An Introduction to Unix for Programmers in the ODU Computer Science Dept.

STEVEN ZEIL

MAR. 21, 2002

A printable version of this document is also available.

Chapter 1

Introduction

This document is designed to introduce students to the basic Unix skills that they will need to work productively on the ODU CS Dept.'s network of Sun workstations and Linux PC's. This is actually the collected lecture notes for CS 252, Introduction to Unix for Programmers, a course offered by the ODU Computer Science Dept. But whether you are enrolled in that course or not, you are welcome to peruse these notes. Occasionally, you will see references to the accompanying textbook for that course, *An Introduction to Unix with X and the Internet*, Paul S Wang, 1997, PWS Publishing.

In addition to those seeking access to the CS Dept.'s Unix network, some people may be interested in the Cygwin project's free port of the GNU C/C++ compilers for use on Windows 95/98/NT/2000 machines. Along with the compilers, this package provides a Unix emulation layer including a Unix-

style command shell called “bash”. Most of what is described in this document applies to working in bash as well. My own notes on installing and using this package are available here.

1.1. Why Unix?

We can actually interpret the title question in a number of different ways. Do we mean, “why does the CS department use Unix?”, or “why is this course about Unix?”, or “why was unix invented?”, or even “why does Unix behave the way that it does?”

It’s actually the last of these interpretations that I want to address, although understanding the answer to that question will go long ways towards explaining why the CS department uses Unix in most of its courses and, therefore, the very reason for the existence of this course.

I think that, to really understand a number of the fundamental behaviors of Unix, it helps to consider how Unix is different from what is probably a much more familiar operating system to most of you, Microsoft Windows. Furthermore, to understand the differences between these operating systems, you need to look at the history of computer hardware and system software evolution in effect at the time when each of these operating systems was designed. In particular, I want to focus on three ideas: evolution of CPU’s and process support, evolution in display technology, and evolution in network and technology.

1.1.1. Mainframe & Minicomputers

CPU and process support

Computer historians are fond of pointing out that mainframe computers were huge behemoths, occupying massive rooms, drawing large amounts of electrical power for their operation, and often requiring cooling systems fully as large as the processor itself. For some time, processors continued to be physically large,

although the processing power squeezed into that space grew tremendously.

On the early machines, only a single program could be run at any given time. As processors became more powerful, both hardware and system software evolved to permit more than one program to run simultaneously on the same processor. This is called multiprocessing. The initial reason for doing multiprocessing was to allow programs from many different users (programmers) to run at once. This, in turn, is called multiprogramming. At first, it was assumed that a single user had no need for more than one process at a time.

Interactive programs are characterized by long periods of idleness, in which they are awaiting the next input from the user. In an interactive environment, it becomes natural for users to switch attention from one process that is awaiting input or, in some cases, conducting a lengthy calculation, to another process that has become more interesting. For example, someone using a word processor might want to switch over to their calendar to look up an important date before returning to the word processor and typing that date into their document. Fortunately, once you have support for multiprogramming, you have most of which you need for combined multiprogramming and multiprocessing.

In fact, there is a definite advantage to having started with multiprogramming. In a multiprogramming environment, there is a great danger that one programmer's buggy software could crash and, by rewriting portions of memory or resetting machine parameters, take down not only that programmer's program but other programs that happened to be running on the machine at the same time. Consequently, multiprogramming systems place a heavy emphasis on security, erecting hardware and software barriers between processes that make it very difficult for one process to affect others in any way.

The trend toward multiprogramming in multiprocessing persisted, not only across families of mainframe computers, but also across the increasing number of desk-size "minicomputers". It is in this con-

text that Unix was developed. From the very beginning, Unix was therefore envisioned as an operating system that would provide support for both multiprocessing and multiprogramming.

Display Technology

During the heyday of the mainframe, most data was entered on punch cards and most output went directly to a printer. Most of these systems had a "console" where commands could be entered directly from a keyboard, and output received directly on an electric typewriter-like printer, but such input was slow and inexact. Prior to multiprocessing, it would have been economic folly to tie up an expensive CPU waiting for someone to type commands and read output at merely human speeds. So the system console generally saw use only for booting up the system, running diagnostics when something was going wrong, or issuing commands to the computer center staff (e.g., "Please mount magnetic tape #4107 on drive 2.").

The advent of multiprocessing and the subsequent rise of interactive computing applications meant that the single system console, hidden away in the computer room where only the computer center staff ever touched it, was replaced with a number of computer terminals accessible to the programmers and data entry staff. An early computer terminal was, basically, a keyboard for input and an electric typewriter for output.

Terminals were not cheap, but their lifetime cost was actually dominated by the amount of paper they consumed. In fairly short order, the typewriter output was replaced by a CRT screen. This opened up new possibilities in output. A CRT screen can be cleared, output to it can be written at different positions on the screen, and portions of the screen can be rewritten without rewriting the entire thing. These things can't be done when you are printing directly onto a roll of paper. Terminal manufacturers

began to add control sequences, combinations of character codes that, instead of printing directly, would instruct the terminal to shift the location where the next characters would appear, to change to bold-face or underlined characters, to clear the screen, etc. All of this wizardry was hard-wired – there were no integrated-circuit CPUs that could be embedded into the box and programmed to produce the desired results. Consequently, terminals were quite expensive (the fancier ones costing as much as a typical new car). Different manufacturers selected their control sequences as much based upon what they could wire in easily as upon any desire for uniformity or compatability. Consequently, there were eventually hundreds of models of CRT-based computer terminals, all of which used incompatible sets of control sequences.

Embedded microprocessors eventually simplified the design of computer terminals considerably (sinking a number of companies along the way that had made their money leasing the older expensive models), and the capabilities of computer terminal began to grow, including adding graphics and color capabilities. Eventually, PCs became cheap enough that the whole idea of a dedicated box serving merely as a terminal came into question, and the computer terminal now exists as a separate entity only in very special circumstances, although there are periodic attempts to revive the idea (e.g., so-called internet appliances).

Network Technology

Before there was a World-Wide Web, there was an Internet. The internet grew out of a deliberate attempt to allow researchers all around the country access to the limited number of highly expensive mainframe CPU's. Internet traffic originally was dominated by telnet, a protocol for issuing text commands to a

computer via the internet, and ftp, a protocol for transferring files from machine to machine via the Internet. Email came along later.

In imitation of (and perhaps in jealousy of) the internet, UseNet evolved as an anarchic collection of mainframe and minicomputers that each knew a handful of telephone numbers of other UseNet computers and could pass email and news (a.k.a. bulletin board) entries along those connections.

As the idea of long range networking took hold, more and more sites began installing local area networks to enable communication among their own machines.

1.1.2. A Tale of Two Operating Systems

Unix

Unix evolved for minicomputers in an historical context where

- Multiprocessing was expected, and the hardware provided safeguards for protecting one running process from affecting or being affected by other processes on the same CPU.
- The most common displays were computer terminals, which came in many different models, all of which used mutually incompatable control sequences. Most of these could display text only, or text with simple vertical & horizontal line graphics. “True” graphics terminals were not unknown, and were clearly on their way, but were so expensive as to be comparatively rare.
- Networking was common. In fact, it was normal, perhaps even the rule, for users to be controlling, via the network, machines that were remote from the users’ actual location.

The PC Revolution

When personal-computer (PC's) came on the scene, they represented a revolution in terms of both decreased size and decreased cost, but they represented a step backwards in terms of total computing power and in terms of the sophistication of the hardware support for many systems programming activities.

Oddly enough, PC systems seemed to recap the entire history of computing up till that time, thoug at a somewhat faster pace:

- "One user – One CPU" was a rallying cry of the early PC proponents. They argued that, although an individual PC presented limited CPU power compared to mainframe or mini machines, the individual PC could still provide a single user with more CPU power than that person would receive as their share of a mainframe when split over a large number of simultaneous users. So early PC operating systems returned to the single-user, single-process[1] model that had gone out of fashion decades before in the world of larger computers.

- Display technology reverted initially to the electric-typewriter style system console. This was quickly supplanted by a "dumb terminal" CRT-and-keyboard, though early printers were still electric typewriter based.[2]

[1] because, after all, that single user didnt really need to split those precious CPU cycles among more than one application at a time, right?

[2] In fact, I recall seeing ads for a device consisting of a panel of solenoids and control circuits that could be placed over the keyboard of an electric typewriter. Send the panel the ASCII code for an "a", and a solenoid "finger" would punch down right where the "a" key would be on a typewriter. Send it the ASCII code for "Z", and a pair of solenoids would strike the typewriter's "shift" and "z" keys.

The very existence of integrated circuit CPUs, however, lowered the cost of CRT displays to the point where more elaborate, graphics-capable displays were soon available.

- Network technology was initially spurned. “One user – One CPU”, remember? Why would anyone need access to other computers. Email and net news could be handled by modem connection without full-fledged networking. It took a surprisingly long time before PC operating systems and applications began to acknowledge that not every bit of information and not every hardware/software resource could economically be replicated on every PC system.

MSDOS & Windows

MSDOS was developed in a context where

- “One user – One CPU” was the rule. Multiple processes for a single user were not deemed necessary.
- Most PCs had a CRT display with limited character and graphics capabilities.
- Networking was deemed unnecessary.

As MSDOS evolved into Windows, it did so in response to changes in the HW/SW context:

- “One user – One CPU” remained the rule, but a single user might have multiple processes.
- PC displays could show characters in a variety of fonts, and graphics capabilities were more common.

- Some people might want local networking, but it was supplied by third-party add-ons with minimal support from the operating system itself. As for the internet, why would anyone with a PC want to communicate with all those mainframe dinosaurs?

1.1.3. Reflections on MS Windows & Unix

Of course, both MS Windows and Unix continued to evolve past their earliest forms, but the contexts in which they have evolved helped establish their fundamental philosophy and continues to influence how they work today.

- Unix users tend to do a lot more typing than Windows users. Graphics capabilities were rare when Unix was developed, so commands had to be typed out rather than always working through a window GUI, and that practice of typing commands continues to influence the "look and feel" of Unix. Unix does have a windowing GUI, but the Unix approach is to launch an application via a typed command, then let that application open up windows if it needs them. Compare to MS Windows, where most users never type a command, and may not even realize that many common Windows applications offer a variety of command line options. (Try, for example, creating shortcuts on your Windows desktop with the Target:

  ```
  C:\WINDOWS\EXPLORER.EXE
  ```

  and another with the Target:

  ```
  C:\WINDOWS\EXPLORER.EXE /n,/e,c:\
  ```

Try them each and see what they do. The difference is potentially useful, but this possibility is unknown to most Windows users.

- Because graphics capabilities were rare when Unix was developed, many Unix applications are text based. The canonical Unix application reads a stream of text in (from "standard input") and produces a stream of text as output (on "standard output"). Because the output is often considered a slightly modified version of the input, such programs are called filters. Unix users are more likely to string together a bunch of filters that each do something simple than to hunt for a massive dedicated application that does the whole job at once. For example, if you wanted to know how many statements occurred in some file of C++ code, an MS Windows programmer would load that file into a visual C++ programming environment, then search through the menu bars for a "properties" or "number of statements" item that might convey the required information. A Unix programmer would, in a single line of text commands, feed that file into a program that extracted each line containing a ";" (because most C++ statements end with semicolons) and then feed that set of ";"-containing lines through a line-counting filter.

- "One user – One CPU" thinking dominates Windows applications even though networking is widely available. MS Windows has never offered the same level of protection against one process affecting others on the same machine. In Windows, it's still all too common for a single crashing application to lock up the machine to the point where a reboot is required. In Unix, such reboots are quite rare.

  In part, this is because Unix allows processes to interact in only two ways:

  – By having one process write out files that the other reads

– By having one process write characters into a “pipeline” that is read by another process.

By contrast, MS Windows has a variety of communication mechanisms, some of which allow one process to directly manipulate the data of another. This allows different applications to work together in interesting and valuable ways (e.g., embedding charts from a spreadsheet directly inside a word-processor document), but at an inevitable cost to overall system security and stability.

- Unix applications are often designed to be run remotely, via a network. Even the Unix windowing GUI, called X, is based on the idea that an application can be run on whatever machine it is installed upon, but will actually display its windows on and accept its mouse clicks and other inputs from any machine on the network. MS Windows, on the other hand, assumes that the application is running on the machine where its outputs should be displayed. If you want to use, say, a paint program that isn’t installed on the PC you are sitting at but is installed on another PC in the same room, you can’t run that program without actually getting up and moving to the other PC.

- Another instance of this bias, one that is, I suspect, much closer to the hearts and pocketbooks of Windows applications developers, can be seen in the deliberate ignorance of many applications to the realities of the networking world. Although Windows will provide support for different users logging in (one at a time!) to given PC, the majority of application software that a user might install on that PC will assume that a single data area and a single set of option and preference settings are enough for that PC. Therefore every time you change a setting, you affect the behavior of that application not only for yourself, but for all other people who use that same machine. Unix applications, on the other hand, are far more likely to be distributed under site licenses permitting

all users at a site equal access.[3]

If you have PCs arranged on a network, so that they can access a common pool of disk drives, when you purchase a new windows application, it is likely to try to install itself onto a single PC, not allowing itself to be run from the other PCs that can access the drive where you install the software. And even if you could run it by from a different PC, you will often find that all of your preferences or options settings and all of your accumulated data is squirreled away in the registry or systems area of the PC on which the software was installed, making it unusable from other PCs.

All this may help to explain why the CS Dept. makes such heavy use of Unix. It's not that we dislike MS WIndows. But if we require a particular software package (say, a compiler) in our classes, under Unix we install it on a few Unix machines and let students run it from remote locations via the internet. Under Windows we have to install it on every CS Dept machine, ask that it be installed on every machine in the ODU laboratories (and at the distance learning sites). And when an updated version of that package comes out, we have to go through the entire process all over again. It's just far, far easier to maintain a consistent working envoronment for all students under Unix.

---

[3]Indeed, Unix applications are far more likely to be distributed with source code. It's worth noting that both the Free Software movement and the Open Source movement originated in the Unix community. By contrast, Microsoft has recently contended that any software that is distributed free of charge or that is distributed with source code is "potentially viral" and "anti-American".

Chapter 2

The Basics

2.1. Logging In

There are two ways to interact with Unix and Unix programs: through a *text-only* interface, or via a *windows* interface — the Unix windowing system is called *X*. Which mode of interaction you use depends upon how you are accessing to the network, the software on the machine you are sitting at, and how fast your connection to the network is. But even if you are running X, one of the first things you are likely to open is an "xterm", a text-only command window. Unix users tend to launch programs from the text-only interface. If the program itself supports windows, mice, etc., then they can point and click to their heart's content.

In this section, we'll concentrate on access via `telnet`, the internet protocol for issuing interactive commands to remote machines. Window-based access will be introduced later.

2.1.1. Making a Connection: telnet

If you are a student registered for a CS course, and have never had a Unix account on the CS Dept system, you can get your account by going to the CS Dept home page and clicking on the "Account Management" link (under "Online Services").

If you have had an account in the recent past, it should be regenerated for you in any semester when you are registered for a CS course. Otherwise, you will need to contact your instructor or from the CS systems staff to get your account.

"`telnet`" is an internet protocol that allows a person connected to the internet to log into other machines on the internet and to issue commands to those machines. To use telnet, you must have a

"telnet client" program on the machine where you are seated, and you must know the name of a machine elsewhere that is running a "telnet server", the program that accepts logins and subsequent commands from the client.

The easiest way to choose a telnet server in our Dept. is to use the name "`fast.cs.odu.edu`", a "fake" machine name that actually requests that you be assigned a random selection among the faster CPU's maintained by the Dept. If you're not into pot-luck, however, you can select a specific machine from this list at `http://www.cs.odu.edu/~public/telnetmachines.html`. Note that you will probably need to add the string "`.cs.odu.edu`" to any of these names.

Next, you need to run a telnet client program. I can't tell you for sure where that will be or how to run it, as that information depends upon the machine you're sitting at. But you're probably looking at one of two situations:

- If you are seated at a CS Dept PC, or at most PC's running some version of the MS Windows operating system, that opeating system already comes with a program named (surprise!) "`telnet`".

  You usually run this by going to the "Start" button, selecting "Run", and entering the command

  ```
  telnet fast.cs.odu.edu
  ```

  (or substitute the name of the specific machine you have chosen for your session).

  Note, however, that Microsoft's `telnet` is really a minimal version. In particular, it does not provide very good keyboard support (certain key combinations that are useful simply can't be typed in using Microsoft's telnet). Therefore, you'll want to look for an alternative ASAP.

- PC's on the ODU Norfolk campus and many, if not all, of the distance sites may have, in addition to the Windows telnet, a program suite called "Hummingbird", which you can locate in the "`Start->Programs` listing as "`Hummingbird Connectivity...->Host Explorer->Telnet`".

  You'll probably find a number of different programs in that group. Select "Default VT" for the terminal type (a VT220) terminal.

- There are a number of free telnet programs that you can download and install on your own machine. See the "Downloads" section on the CS 252 Library page for some recommendations.

2.1.2. Logging In

Now that you have a login prompt, enter your login name. At the "password:" prompt, enter your password.

After a few moments, you should receive a command prompt.

2.1.3. Setting Your Terminal Type

Remember that Unix evolved in a time when many manufacturers made many different models of computer terminals, each with its own set of command codes for clearing the screen, moving the cursor to different screen positions, setting bold face, underline and other text characteristics, and so on. A typical Unix installation will be equipped to communicate with any of a few hundred different types of terminals.

Now, you're not using a terminal, but you are using a telnet program. `telnet` was originally intended to allow terminals to issue commands to remote CPUs, so your telnet client program actually works by simulating an "old-fashioned" computer terminal. The authors of your telnet program chose one or more kinds of terminal that they would simulate. For Unix to manipulate your screen appropriately, it most know what kind of terminal command codes your telnet program is prepared to accept.

Find out the kind of terminal being emulated by your telnet program. You may need to consult the program documentation or help files for this. You may also be able to deduce this information from the program's "Options" or "Preferences" menus. Some programs may give you a choice of several terminals. Common choices include "vt100", "vt102", "vt52", "vpoint", and "adm3a". The "vt10?" series is usually one of your better bets, but you may want to try different ones to see which works best for you.

Now, there is a protocol by which Unix systems can ask a terminal, "What kind are you?", and the terminal responds with an identification code. Many simpler telnet clients don't implement this feature, however, so you can't take it for granted.

Look at the messages that you received after after logging in. If it includes a line saying something like "Terminal type is…" and the terminal type that it names makes sense for your telnet program, you're all set and you can skip the next step. But it there terminal type seems wrong, or you see a message indicating that you are on a "dumb terminal"[1], you need to tell the Unix system what kind of terminal you are really emulating. The command to do so is

```
setenv term  xxxx
```

[1] A dumb terminal is one that displays lines of text but has no command codes for moving things to specific locations on the screen or doing other basic operations. Many Unix applications, such as text editors, will not work with dumb terminals.

where *xxxx* is the kind of terminal (e.g, `setenv term vt100`).

Some people also recommend that you follow this command with

```
tset -Q
```

which resets the terminal. In my own experience, this is usually unnecessary,and I have found that many communications programs don't deal well with this, but try it if your terminal seems to be misbehaving.

Most "dumb" terminals provide for 24 lines of text. Many telnet programs, however, allow more. If yours is one of these, you should tell Unix how many lines you are using by giving the command

```
stty rows  nn
```

where *nn* is the number of rows/lines.

2.1.4. Changing Your Password

Whether we like it or not, we need to worry about the security of our computing environment. There are people who would take advantage of this computer system if they had any, or more complete, access to it. This could range from the use of computer resources they have no right to, to the willful destruction and/or appropriation of the information we all have online. In order to maintain the level of security in our computing environment that we need, there are some things we all have to take responsibility for. Even though you may not feel like you personally have much to lose if someone had access to your account or files, you have to realize that as soon as someone gains ANY access to our system, it's 100 times easier for them to gain access to ALL of it. So when you are lax with your own account, you are endangering the work and research of everyone else working here.

Your password is the fundamental element of security not only for your personal account, but for the whole UNIX system that we share. Without an account and password a person has NO access to our system. If someone discovers (or you tell someone) your password, not only will they have access to your personal files, but they will have a much better chance to launch attacks against the security of the entire system.

Your account password is the key to accessing and modifying all of your files. If another user discovers your password, he or she can delete all your files, modify important data, read your private correspondence, and send mail out in your name. You can lose much time and effort recovering from such an attack. If you practice the following suggestions, you can minimize the risk.

1. **NEVER** give another user your password. There is no reason to do this. You can change permissions and have groups set up if you need to share access with other individuals. Your account should be yours alone.

2. Never write down your password. Another person can read it from your blotter, calendar, etc. as easily as you can.

3. Never use passwords that can be easily guessed. Personal information about you (birth date, etc.) may be known to the attacker or may be recorded in on-line databases that the attacker has already obtained.

   Passwords should not be single words (in any language) because on-line dictionaries are widely available for use in spelling checkers. A common approach to cracking passwords is to compile a set of such words and to run a program that tries each one on each account on the machine. Consider inserting punctuation and other “odd” characters into your password to foil such attacks.

A person with local knowledge can also try your spouse's name, pets' names, etc. Your account is vulnerable to this type of cracking unless you choose your password carefully.

4. Change your password the very first time you log in, and every few months thereafter. Security problems are often traceable to stale passwords and accounts. These are accounts that have become inactive for one reason or another or the password has not changed for a long time. In our particular environment we have had break-ins via such stale accounts. A password that remains the same for a long time provides an intruder the opportunity to run much more advanced and longer running programs to break such passwords.
5. Vary the system by which you choose a password. For example, don't repeatedly use combinations like BLUEgreen and REDyellow. If an intruder discovers your pattern, he or she can guess future passwords.

The command to change your password is

```
passwd
```

This command will first prompt you for your old password (just to check that you really are you!) and then will ask you to type your new password (twice, so that an inadvertent typing mistake won't leave you with a password that even you don't know!).

2.1.5. Logging Out

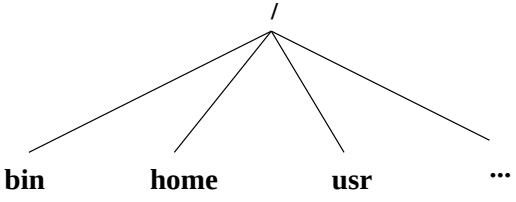
To leave our Unix systems, type

```
exit
```

(not `logout`, as indicated in the textbook.)

2.2. The Unix File System

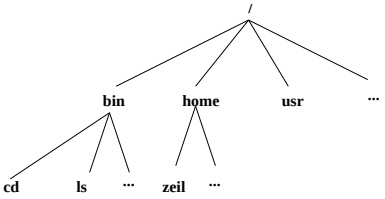
Files in Unix are organized by listing them in *directories*. Directories are themselves files, and so may appear within other directories. The result is a tree-like hierarchy. At the root of this tree is a directory known simply as "/".[2] This directory lists various others:



The `bin` directory contains many of the programs for performing common Unix commands. The `usr` directory contains many of the data files that are required by those and other commands. Of particular interest, however, is the `home` directory, which contains all of the files associated with individual users like you and me. Each individual user gets a directory within `home` bearing their own login name. My login name is `zeil`.

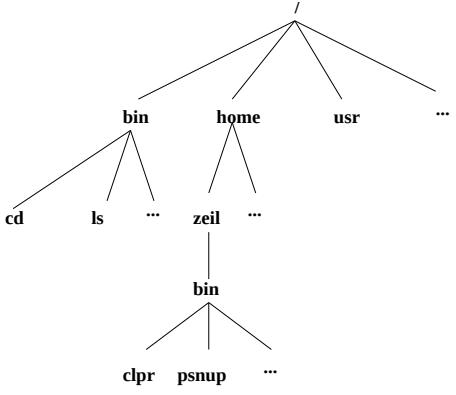
We can expand our view of the Unix files then as:

[2] It may be more precise to say that this directory's name is the empty string "".



`cd` and `ls` are two common Unix commands, as will be explained later.

Within my own home directory, I have a directory also named “bin”, containing my own personal programs. Two of these are called “clpr” and “psnup”. So these files are arranged as:



The full name of any file is given by listing the entire path from the root of the directory tree down to the file itself, with “/” characters separating each directory from what follows. For example, the full

names of the four programs in the above diagram are

```
/bin/cd
/bin/ls
/home/zeil/bin/clpr
/home/zeil/bin/psnup
```

There are some common abbreviations that can be used to shorten file names.

- You can refer to the home directory of someone with login name *name* as `~name`.
- You can refer to your own home directory simply as `~`.

  So you could refer to the file containing my `clpr` program as either `/home/zeil/bin/clpr` or `~zeil/bin/clpr`.

  When I myself am logged in, I can refer to this program by either of those two names, or simply as `~/bin/clpr`.
- At all times when entering Unix commands, you have a "working" directory. If the file you want is within that directory (or within other directories contained in the working directory), the name of the working directory may be omitted from the start of the filename. When you first log in, your home directory is your working directory. For example, when I have just logged in, I could refer to my program simply as `bin/clpr`, dropping the leading `/home/zeil/` because that would be my working directory at that time.
- The working directory itself can be referred to as simply ".".

- The “parent” of the working directory (i.e., the directory containing the working directory) can be referred to as “..”.

Unix filenames can be almost any length and may contain almost any characters. As a practical matter, however, you should avoid using punctuation characters other than the hyphen, the underscore, and the period. Also, avoid blanks, and non-printable characters within file names. All of these have special meanings when you are typing commands and so would be very hard to enter within a filename.

Some things to keep in mind about Unix file names that may be different from other file systems you have used:

- Unix file names are often very long so that they describe their contents.[3] The rather perverse exception to this rule is that program/command names are, by tradition, very short, often confusingly so.
- Upper and lower case letters are distinct in Unix filenames. “`MyFile`” and “`myfile`” are different names.
- Periods (“.”) are not treated by Unix as a special character. “`This.Is.a.legal.name`” is perfectly acceptable as a Unix filename. Many programs, however, expect names of their data files to end in a period followed by a short “standard” extension indicating the type of data in that file. Thus data files with names like “`arglebargle.txt`” for text files or “`nonsense.p`” for Pascal source code are common.

[3] As we will see, one almost never needs to type an entire filename in a Unix command, so long file names are no harder to work with than short ones.

By convention, files containing executable programs generally do not receive such an extension.

- Keep in mind that directories are separated by “/” in file names, not by “\” as is common in some other operating systems.

2.3. Typing Unix Commands

To run a Unix command (or any program, for that matter), you normally must type the name of the command/program file followed by any arguments. There is actually a program running that accepts your keystrokes and launches the appropriate program. The program that reads and interprets your keystrokes is called the *shell*. There are many shells available, all of which offer different features. The default shell for ODU CS is called tcsh, and we'll concentrate on that.

The command/program name is usually not given as a full file name. Instead, certain directories, such as `/bin`, are automatically searched for a program of the appropriate name. Thus, one can invoke the `ls` command as either

```
/bin/ls
```

or simply as

```
ls
```

As you type, some characters have a special meaning. For example, if you have entered the first few letters of a file name and hit the "Tab" key, the shell will examine what you have typed so far and attempt to complete the filename by filling in the remaining characters. If the shell is unable to complete the filename, a bell or beep sound will be given. Even in this case, the shell will fill in as many characters as it can.

Most special characters are entered by holding down the "Control" key while typing a letter. By convention, we designate this by placing the symbol "^" in front of the name of the second key. For

example, if you have typed zero or more letters of a filename and want to see a list of what filenames begin with what you have typed, you could type ^D, i.e., hold down the "Control" key and type "d".

Some other useful special keys are:

- ^C is used to abort a program/command that is running too long or working incorrectly. Beware: aborting a program that is updating a file may leave garbage in that file.
- ^D is used when a command/program is reading many lines of input from the keyboard and you want to signal the end of the input.
- ^H, the "Backspace" key, and the "Delete" key all delete the most recently typed character.
- ^B moves the cursor **B**ackwards over what you have just typed, without deleting those characters. This is useful in correcting typing mistakes. The "left" arrow on your keyboard may also do the same thing.
- ^F moves the cursor **F**orwards over what you have just typed, without deleting those characters. The "right" arrow on your keyboard may also do the same thing.
- ^P retrieves the **P**revious command that you had typed in. Repeated ^P's may be used to look back through a number of commands that you have issued recently.

  The "up" arrow on your keyboard may also do the same thing.
- ^N is the opposite of ^P. After one or more ^P's, a ^N allows you to move back to the **N**ext more recent command.

  The "down" arrow on your keyboard may also do the same thing.

- In many programs, ^Z pauses the program and returns you temporarily to the shell. To return to the paused program, give the command: `fg`

2.4. Some Basic Unix Commands

If you have not yet done so, log in now so that you can work though the following commands.

Upon logging in, your *working directory* should be your home directory. The command `pwd` will **p**rint the **w**orking **d**irectory. Give the command

```
pwd
```

You should see something like

~yourname

Now, let's make a place to play in. `mkdir` will make a new directory. Enter the command

```
mkdir playing
```

to create a directory named "playing".

The command `ls` lists the contents of the working directory. More generally, `ls` *`directoryname`* will list the contents of any directory.[4]

Give the command

```
ls
```

[4] Well, not really *any* directory. People can protect their own directories from the prying eyes of others, in which case `ls` will fail.

and you should see `playing` listed. In fact, it may be the only thing listed.

The command `cd` is used to **c**hange the working **d**irectory. Give the command sequence

```
pwd
cd playing
pwd
cd ..
pwd
cd ./playing
pwd
```

to see this in action.

The `cp` command copies one or more files. You can give this command as `cp file1 file2` to make a copy of file `file1`, the copy being named `file2`. Alternatively, you can copy one or more files into a directory by giving the command as

```
cp file$_1$ file$_2$... file$_n$ directory
```

Now try the following:

```
ls ~public/Misc              [Notice that there are a number
                              of files ending with .txt]
cp ~public/Misc/*.txt ~/playing
ls ~/playing
```

The "`*`" is a *wildcard* character. It tells the shell to substitute any combination of zero or more characters that results in an existing filename. In cases where there are multiple possibilities, such as this one, the shell forms a list of all the matches. So the `cp` command actually saw a list of all files in the `~public/Misc` directory whose names ended in ".txt".

To get a better feel for wildcards, try the following:

```
ls /usr/include
echo /usr/include
echo /usr/include/*.*            [List only file names containing a "."]
echo /usr/include/*
echo /usr/include/f*.*
echo /usr/include/*f*.*
```

(The `echo` command simply lists its arguments on the screen for you to see, after doing any special processing such as wildcards.)

Here are some other common Unix commands. Try experimenting with these in your `~playing` directory.

cancel *request* Remove a file from the printer queue, so that it won't get printed. The *request* identifier is found using the `lpstat` command.

cat *file*$_1$...*file*$_n$ Lists the contents of each of the listed files on your screen.

exit Shut down the current shell. If this shell is the one you got at log-in, this command logs you out.

mv *file*$_1$ *file*$_2$ Renames *file*$_1$ as *file*$_2$. *file*$_2$ may be in a different directory.

mv *file*$_1$ *directory* Moves *file*$_1$ to the given directory.

lp *file* Send file to printer for printing. Most sites have multiple printers, each having its own name. One of these will be the "default" printer used by the `lp` command. For the others, you must give the printer name as part of the printer command:

```
lp -d  printer  file
```

For example, at the Norfolk ODU campus, `lp` by itself prints to a fast printer in the public workstation lab for text only. `lp -dcookie` prints to "cookie", a printer in the same room that offers the extra capability of printing Postscript graphics.

You will need to consult your local staff to see what printers are available at other sites.

lpstat Shows the list of files you have "queued up" awaiting their turn on printers. Entered by itself,

```
lpstat
```

it lists only your own print jobs, giving a unique identifier for each one.

To see the entire list of print jobs on some printer, enter

```
lpstat -o  printer
```

ls -a By default, filenames beginning with "." are considered "hidden" and not shown by the `ls` command. The `-a` (for "**a**ll") option causes these files to be shown as well.

ls -l This is the "long" form of `ls`. It displays additional information about each file, such as the size and date on which the file was last modified.

Note that options can be combined. For example, you can say `ls -la` to get extra information including normally hidden files.

ls -F Adds a little bit of extra information to each file name. If the file is an executable program, its name is marked with an "*". If the file is a directory, its name is marked with "/".

more ***file***$_1$**...*** **file***$_n$ Lists files one screen at a time, pausing after each screen-full. Hit the space bar to advance to the next screen.

A related program is `less`, which also allows you to move backwards through the files by hitting "b".

rlogin ***machine*** Logs you in to another machine on the network. Use this if the machine you are on seems to be running slowly and the `who` command indicates that there are lots of others on the same machine.

rm ***file***$_1$**...*** **file***$_n$ Deletes the listed files. Be very careful using wildcards with this command. `rm *` will delete everything in the current working directory!

rm -i ***file***$_1$**...*** **file***$_n$ Deletes the listed files, but firsts asks permission to delete each one.

rm -r ***file***$_1$**...*** ***file***$_n$ Deletes the listed files. If any of these files is a directory, it deletes that directory and everything in it as well.

rmdir ***directory*** Deletes a directory.

who Lists everyone logged into the same machine that you are using.

2.5. Redirection and Pipes

One of the interesting ideas that pervades Unix is that many, if not most, programs can be viewed as "filters" or "transforms" that take a stream of text as input and produce an altered stream of text as output. Many Unix commands are designed to perform relatively trivial tasks, perhaps not very useful by themselves, that can be chained together in interesting and useful ways.

The practical consequence of this is that Unix shells devote special attention to a *standard input* stream that forms the main input to most programs/commands, and to a *standard output* stream that forms the main output from most programs/commands.[5] The shell attempts to make it easy either to *redirect* one of these standard streams to a file or to *pipe* the standard output stream of one program into the standard input of another.

For example, the program `wc` (for **w**ord **c**ount) reads text from its input stream and produces as its output stream three number indicating the number of lines, words, and characters that it saw. You could invoke this directly:

```
wc
Hello.
How are you?
^D
```

in which case, you would see as output:

[5]There is actually a second output stream supported by many programs, the *standard error* stream, used for writing error/debugging messages.

```
      2      4     20
```

For this to be very useful, however, we need to make it accept a file as input. This is done by using the `<` operator in the shell. Think of the `<` as an arrow indicating data flowing towards the command from a filename: If `hello.c` is this file:

```
#include <stdio.h>
int main ()
{
  printf ("Hello from C!\n");
  return 0;
}
```

then the command

```
wc < hello.c
```

produces the output

```
      6     13     80
```

On the output end, the shell operator `>` directs the standard output into a file (again, think of this as an arrow indicating data flowing into a filename from the command):

```
wc < hello.c > hello.wc
```

produces no output on the screen, but creates a file called `hello.wc`. That file will contain the output

```
    6    13    80
```

of the `wc` command.

The output redirection operator has a couple of important variants. First, the shell generally does not allow you to redirect into an existing file. If you give the command

```
wc < hello.c > hello.wc
```

a second time, the shell will refuse to perform the command. You can force the shell to delete an existing file and create a new one for redirection by changing the `>` to `>!`.

Second, sometimes we would like to add output to the end of an existing file instead of replacing that file. This is done with the operator `>>`. So the code sequence

```
wc < hello.c >! hello.wc
wc < hello.c >> hello.wc
```

would result in a file `hello.wc` with contents

```
    6    13    80

    6    13    80
```

regardless of whether `hello.wc` had existed previously.

To pipe the output of one command into the input of another, use the shell operator `|`. A common example of a pipe is to take a command that may have a large amount of output and to pipe it through `more` to facilitate viewing. For example, try

```
ls /bin | more
```

As you gain facility with a greater variety of Unix text manipulation commands, you will find that redirection and pipes can be a powerful combination. For example, suppose that you have written program `myprog` that emits a great deal of output, among which might be some error messages starting with the phrase "`ERROR:`". If you wanted to read only the error messages, you could, of course, just view *all* the output, watching for the occasional error message:

```
myprog | more
```

But if the program produces a lot of output, this will quickly become tedious. However, the program `grep` searches its input stream for a given character string,[6] emitting at its output only the lines containing that string. By piping through `grep`, we can limit the output to the part we really want to see:

```
myprog | grep "ERROR:" | more
```

---

[6] Actually, `grep` is far more powerful, enabling you to search for strings matching elaborate patterns.

2.6. File Protection

2.6.1. Protections

Not every file on the system should be readable by everyone. Likewise, some files that everyone needs (such as the executables for commands like `cp`, `mv`, etc.) should not be subject to accidental deletion or alteration by ordinary users. This is where file *protection* comes into play.

Unix allows three forms of access to any file: read, write, and execute. For an ordinary file, if you have read (r) permission, you can use that file as input to any command/program. If you have write (w) permission, you can make changes to that file. If you have execute (x) permission, you can ask the shell to run that file as a program.

The owner of a file can decide to give any, all, or none of these permissions to each of three classes of people:

- To the owner of the file him/herself
- To members of a designated "group" established by the systems staff. Groups are generally set up for people who will be working together on a project and need to share files among the group members.
- To anyone else in the world.

These three classes are abbreviated "u", "g", and "o", respectively. The "u" is for "**u**ser", "g" for "**g**roup", and "o" is for "**o**thers". Until you actually join a project that needs its own group, you will mainly be concerned with "u" and "o" classes.

The `ls -l` command will show the permissions granted to each class. For example, if you said

```
ls -l ~/playing
```

you might see the response

```
-rwxrwx---  1 zeil       311296 Jul 21 09:17 a.out
-rw-rw----  1 zeil           82 Jul 21 09:12 hello.c
-rw-rw----  1 zeil           92 Jul 21 09:13 hello.cpp
-rw-rw----  1 zeil           85 Jul 20 15:27 hello.wc
```

Look at the pattern of hyphens and letters at the left. The first character will be a "d" if the file is a directory, "-" if it is not. Obviously, none of these are directories. The next 3 positions indicate the owner's (u) permissions. By default, you get read and write permission for your own files, so each file has an "r" and a "w". `a.out` is an executable program, so the compiler makes sure that you get execute (x) permission on it. The other files can't be executed, so they get no "x". This way the shell will not even try to let you use `hello.c` or any of the other source code files as a program.

The next three character positions indicate the group permissions. In this case, the group permissions are the same as the owner's permissions. Knowing the group permissions isn't very helpful, however, unless you know to which group it refers. The command `ls -g` produces output similar to `ls -l` but lists the files' groups instead of their owners. For example, if you said

```
ls -l ~/playing
```

you might see the response

```
-rwxrwx---  1 student    311296 Jul 21 09:17 a.out
-rw-rw----  1 student        82 Jul 21 09:12 hello.c
-rw-rw----  1 student        92 Jul 21 09:13 hello.cpp
-rw-rw----  1 student        85 Jul 20 15:27 hello.wc
```

Some typical groups are “wheel”, “faculty”, “gradstud”, and “student”. “Wheel” has no members, but groups like “student” and “gradstud” have very broad membership, as their names imply. Although, as we shall see, the files in this example do not give any privileges to the world (others), they can be read and written by all students because of the group permissions.

The final three character positions indicate the permissions given to the world (others). Note that in this case, people other than the owner or members of the same group cannot read, write, or execute any of these files.

Directories also can get the same `rwx` permissions, though the meaning is slightly different. If you have read permission on a directory, you can see the list of files in the directory via `ls` or other commands. If you have execute permission on a directory, then you can use that directory inside a file name to get at the files it contains. So, if you have execute permission but not read permission on a directory, you can use those files in the directory whose names you already know, but you cannot look to see what other files are in there. If you have write permission on a directory, you can change the contents of that directory (i.e., you can add or delete files).

2.6.2. chmod

The `chmod` command changes the permissions on files. The general pattern is

`chmod` *`class`*± *`permissions`* *`files`*

Use "`+`" to add a permission, "`-`" to remove it. For example, `chmod o+x a.out` gives everyone permission to execute `a.out`. `chmod g-rwx hello.*` denies members of your group permission to do anything at all with the "hello" program source code files.

You can also add a `-r` option to `chmod` to make it "recursive" (i.e., when applied to any directories, it also applies to all files in the directory (and if any of those are directories, to the files inside them, and if…). For example, if I discovered that I really did not want the group to have permission to write or execute my files in `~/playing`, I could say:

```
chmod -r g-rx ~/playing
```

2.6.3. Beware the umask!

Suppose you never use the `chmod` command. What would be the protection levels on any files you created?

The answer depends upon the value of `umask`. Look in your `~/.cshrc` file for a command by that name, and note the number that follows it. If you don't have one, just give the command

```
umask
```

and note the number that it prints.

The umask number is a 3 digit (base 8) number. The first digit describes the default permissions for the owner (you), the second digit describes the default permissions for the group,[7] and the final digit

[7] Of course, if the number appears written using only 1 or 2 digits, the missing digits are simply leading zeros.

describes the default permissions for others.

Each of these three numbers is, in turn, formed as a 3-digit binary number where the first digit is the read permission, the second is the write permission, and the thrid digit is the execute permission. In each binary digit, a 0 indicates that the permission is given, a 1 that the permission is denied.

So if my umask is 027, that means that

- I (the owner) have 000 — permission to read, write and execute my own files.
- The group to which a file belongs has 010, permission to read, no permission to write, and permission to execute that file.
- The rest of the world has 111, no permission to read, write or execute.

Of course, these permissions can be changed for individual files via the `chmod` command. The umask only sets the default permissions for cases where you don't say `chmod`.

If you want to change your default permissions, you do it via the `umask` command by giving it the appropriate 3-digit octal number for the new default permissions. Some common forms are:

`umask 022` Owner has all permissions. Everyone else can read and execute, but not write.

`umask 077` Owner has all permissions. Everyone else is prohibited from reading, writing, or executing.

Since the point of the `umask` command is to establish the default behavior for all your files, this command is normally placed within your `.cshrc` file.

2.6.4. Planning for Protection

At the very least, you will want to make sure that files that you are preparing to turn in for class assignments are protected from prying eyes. You need to do a little bit of planning to prepare for this. There are two plausible approaches:

- Use a stringent enough umask (e.g., `umask 077`) so that everything is protected by default.
  - The only disadvantage is that files that you *want* to share (e.g., the files that make up your personal Web page) must be explicitly made world-readable (`chmod go+r files`).
- Use a more relaxed umask (e.g., `umask 022`) so that your files are readable by default, but establish certain directories in which you carry out all your private work and protect those directories so that no one can access the files within them. For example, you might do

  ```
  cd ~
  mkdir Assignments
  chmod go-rwx Assignments
  ```

  Now you can put anything you want inside `~/Assignments`, including subdirectories for specific courses, specific projects, etc. Even if the files inside `~/Assignments` are themselves unprotected, other people will be unable to get into `~/Assignments` to get at those files.
  - The one disadvantage to this approach is that it calls for discipline on your part. If you forget, and place your private files in another directory outside of `~/Assignments`, then the relaxed umask means that those files will be readable by everyone!

2.7. Getting Help

As you explore Unix, you are bound to have questions. Some ways to get answers include:

- The entire Unix manual is on-line.

  `man` *`command`*

  displays the manual page for the given command.

  `man -k` *`keyword`*

  looks up the given keyword in an index and lists the commands that may be relevant.

- The CS Department systems staff has collected a variety of additional help documents. You can find them by going to the Dept home page (`http://www.cs.odu.edu/`) and selecting "General Help".

  and then selecting "Unix & Labs".

- A staff member is generally on duty in the public CS workstation room of the Norfolk campus whenever that room is open.

- If none of the above help, then send e-mail to "root". This is also how you report bugs, machine failures, etc.

Chapter 3

Editing Files

3.1. Editing Text Files

An editor is a program that allows you to easily create and alter text files. There are a variety of editors on the system, of which the most popular are `vi` and `emacs`. Neither is exactly the easiest thing in the world to learn to use. I suggest learning `emacs`, because

- As you gain more facility with `emacs` and with Unix in general, you will find that `emacs` offers many advanced facilities for doing specific tasks.
- In particular, `emacs` offers a number of aids for programmers, including syntax-based highlighting of different programming languages (using different colors/fonts for reserved words, comments, strings, etc.), commands for checking matching {}, for compiling programs and collecting the error messages, and for stepping through those error messages, and for debugging the compiled code.
- `emacs` is widely available (for free) for all Unix systems and also for MS Windows.

To run `emacs`, make sure that you have correctly identified your terminal kind. Then give the command

```
emacs
```

Then follow the directions given to bring up the tutorial (i.e., type ^h followed by "t".).

When you are done with the tutorial, here are few extra things you should know about `emacs`:

3.1.1. In Search of Meta

The tutorial describes a number of commands like this: `M-<chr>`, and says the "`M-`" means to "hold the META or EDIT or ALT key down while typing" the other character, and says that if you don't have a META, EDIT, or ALT key that you can type (and release) the Escape key instead.

It's worth noting that you may, indeed, have some of those keys on your keyboard, but your telnet program may not pass them on. Some telnet programs may jsut ignroe those keys. In other cases (especially the ALT keys), the Windows operating system may be jealously seizing upon those keys for its own purposes. In some cases, you may have 2 ALT keys, one on the left of the spacebar and one on the right. Sometimes one of these will work and the other will not, so try them both. But if nothing else works, the Escape key will always be there for you. (If you change telnet programs, though, or if you start using X to connect as we will learn later, you might want to check out those keys once again.)

3.1.2. The .emacs Startup File

Before starting up, `emacs` tries to read a file `~/.emacs` Many people store special commands in there to customize `emacs` to their own liking. (Reading the `.emacs` file of an experience `emacs` user can be instructive although, unfortunately, sometimes a bit intimidating. (Feel free to take a look at mine.)

If you don't already have a `~/.emacs` file, you might want to start by copying mine this way:

```
cp ~zeil/public_html/software/.emacs ~
```

After you gain some basic proficiency in `emacs`, you may want to take a good look at that file to see what's in there and to adjust it to your own liking.

3.1.3. Emacs Modes

`emacs` offers a customized modes for different kinds of files that you might be editing. Some of these are chosen automatically for you depending upon the file name you give. Others can be chosen automatically by giving the command `M-x` *`name`*`-mode` where *name* indicates the desired mode. Some of the most popular modes are: `text`, `pascal`, `c`, and `c++`. The programming language modes generally offer automatic indenting at the end of each line, though you may have to end lines with the "Line feed" or "C-j" key rather than "Return" or "Enter" to get this.

Try this: Start up emacs and visit a C++ file from one of your other classes (or create a new one, typing in a few lines of C++ code). At the very least, you should see a "(C++)" indicator in the status line indicating that you are in C++ mode. In this mode, you have available special commands for compilation and debugging (which we will discuss in a later lesson). If your terminal program supports color and you copied my .emacs file as suggested earlier, you may see colored "syntax highlighting" as emacs renders reserved words, string constants, and comments in different colors and/or fonts.

And there are little, more subtle things that you will come to appreciate over time. For example, each time you type a ")" or "}" in your code, the cursor briefly flashes back to the "(" or "{" that you just closed.

The command `M-/` is a special friend to all programmers who use long variable names but hate to type them. Type a few letters of any word, then hit `M-/`. `emacs` will search backwards through what you have previously typed looking for a word beginning with those letters. When it finds one, it fills in the remaining letters. If that wasn't the word you wanted, just hit `M-/` again and `emacs` will search for a different word beginning with the same characters.

3.1.4. The Mark and the Region

emacs has a number of commands that work on an entire block of text at a time. For example, the emacs tutorial told you how to delete a line using C-k. But what if you wanted to delete everything from the middle of one line to the 1st word five lines away? There is a command (C-w) for killing an entire region of text, but to use it you must first tell emacs what region you want to kill.

The procedure for doing this is the same in all emacs commands that work on regions of text. The "current region" is the set of characters from the "mark" to the current cursor position. The "mark" is an imaginary position marker established by the set-mark-command. The keystrokes for that command are either C-[spc] (hold the control key and type a space) or C-@ (hold the control and shift keys and type '2').[1]

So to set up a region to operate on, you move the cursor to one end of the region, give the set-

[1]Some telnet programs (notably, the MS Windows built-in telnet) will not let you type the characters C-[spc] or C-@. These characters are the emacs keys used to set the "mark" when selecting a region of text.

You can always run the set-mark command as M-x set-mark-command (but who really wants to type all that every time?)

You could just get a different telnet program. There are a number of good freeware telnet clients available. See the "Downloads" section on the CS 252 Library page for some recommendations.

You can bind a different key to set-mark-command that you can type from within Windows telnet. Add the line:

```
(global-set-key "\M-\\" 'set-mark-command)
```

to your .emacs file, restart emacs, and you can set the mark with the key sequence M-\ (escape followed by a backslash)

I've added that global-set-key to my own .emacs file, so if you've copied that into your home directory as described earlier, that key binding will already be defined.

`mark-command`, then move the cursor to the other end of the region. Everything between the mark and the cursor constitutes the current region, and can be operated on by any region-based command. Some region commands of note are:

`C-w` Kill the region, i.e., delete it but save the deleted text in the clipboard. You can "yank" (paste) the deleted text at the cursor position with `C-y`.

`M-w` Copy the region, i.e., Save a copy of the region's text in the clipboard. You can "yank" (paste) the deleted text at the cursor position with `C-y`.

`C-cC-c` Comment out a region - in C++ and other programming modes, places comment markers in front of each line in the region.

`M-x spell-region` Run a spell check on all text in the region.

3.1.5. Where's the Documentation?

It's in emacs. The end of the tutorial discussed some of the builtin help features in emacs. One that isn't mentioned is the way to get to the entire "reference manual" for emacs. The "info viewer" gives you access to extensive documents about emacs (and about a number of other programs as well, as authors of many other programmers have found the info viewer a convenient way to package on-line documentation.) The commands `C-h i` or `M-x info` will launch the info viewer, and the first page of the viewer gives basic instructions on how to use it.

3.1.6. For the emacs-phobes

Finally, I will note that, if you develop an incurable allergy to emacs, there are other editors that offer reasonable support to programmers. The textbook devotes an entire chapter to `vi`, which does not offer much support for programming, but a reasonable option is `vim` ("vi improved").

Like emacs, vim is available on many, but not all, Unix systems and, once you have learned it, you can use it over telnet via keyboard commands. To learn the basics of running vim, give the command `vimtutor`.

Chapter 4

The X Windowing System

4.1. The X Window System

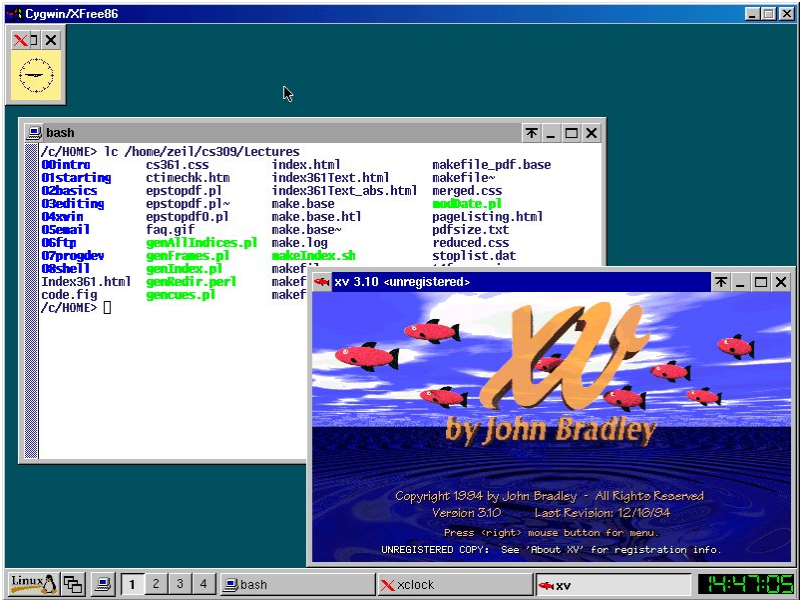
X is the windowing system for Unix. By running X, you can have several windows on the screen open at once, each devoted to a different task. For example, you can be reading electronic mail in one window while a lengthy compilation is running in another. X also allows the display of graphics and of a variety of fonts.

4.1.1. X Window Managers

X is a windowing system that can present a number of different appearances to the user. The appearance and behaviors that you actually see is controlled by a *window manager*, a program that is generally run as part of the X start-up procedure. A window manager controls both cosmetic details like the colors used for window borders and other control elements, but also what elements actualy appear on each window (e.g., does every window get a “close” button, what does that button look like, and where does it appear in the window?) and what menus appear in response to mouse clicks in various parts of the display. Figure 4.1, for example, shows X running the `twin` (Tom’s Window manager). Compare to Figure 4.2, which shows the same programs running under the `ICE` window manager and Figure 4.3, which shows those programs running under a version of X that uses MS Windows as its window manager. You can see a number of different window managers in action here.

(full size)

Figure 4.1: X with the TWM window manager



(full size)

Figure 4.2: X with the ICE window manager

(full size)

Figure 4.3: X with the XWin32 window manager

4.1.2. Running X

To use X, you need to run two different programs. One is an X server, the program that is responsible for controlling the display of the machine on which you are working. The other is an X client, which is simply an application program that was compiled to work with X.

When the X client is running, it draws its windows and graphics using calls to the X libraries. These libraries actually send the appropriate drawing commands on to the server software, often on a different machine. How does the client know to which machine the drawing commands should be sent? The client is told this, when it is run, via the DISPLAY value. An X DISPLAY value looks like:

```
serverMachineName:DisplayNumber
```

The `DisplayNumber` is numeric, and is usually "0" unless you have multiple screens on the same machine (or are working around an NAT system - see 4.1.4).

The serverMachineName is the internet DNS name (e.g., daffodil.cs.odu.edu) or IP address (e.g., 127.0.0.1) of the machine you are on. If you are using a machine in an ODU lab, it probably has a fixed name. If you are working through your own Internet Service Provider, it is possible that your name and IP address change every time you connect. If you aren't sure what to use here, here are some ways to find out:

- If you are running X-Win32, start the X-Util32 program, select "`Options->Display`" from the menu, and you will see what IP addresses X-Win32 thinks are available. In fact, X-Win32 usually handles the DISPLAY setting for you, so you may not even need to know this information.

- Select the “Start” button on the Windows task bar, then “Run”. For the program to open, type “`winipcfg`”. You may need to select among several adapters to find the one you are using for your current connection, but, having selected it, you will see your current IP address.

There are two ways that an internet client can be given the DISPLAY value.

- First, you can set this as an environment variable in the shell from which you are going to launch the client. Try entering

  ```
  echo $DISPLAY
  ```

  from a Unix or Cygwin shell, or

  ```
  echo %DISPLAY%
  ```

  from an MSDOS prompt. If you don’t see an appropriate DISPLAY value already defined, you can define it like this

  ```
  setenv DISPLAY serverMachineName:DisplayNumber
  ```

  for a Unix machine running C-shell or TC-shell,

  ```
  export DISPLAY=serverMachineName:DisplayNumber
  ```

  in CygWin, or

```
set DISPLAY=serverMachineName:DisplayNumber
```

in an MSDOS prompt. Thereafter, any X client program you launch from that shell will know the DISPLAY value you have set.

- Alternatively, you can pass that value as a command line parameter when the client is launched. This almost always takes the form

```
-display serverMachineName:DisplayNumber
```

Compare this, for example, to the instructions for creating an X-Win32 session in the upcoming 12, and you can see the `-display` flag in use. In this case, the display value is just `$DISPLAY`, which is a value filled in by the X-Win32 software.

With this information in hand, the first thing you need to do is to run an X server on the machine where you are actually seated. How you do this depends on what X server has been installed on that machine. I'll assume for the purposes of this dicussion that you are working from a Windows PC.

ODU machines usually have StarNet's X-Win32 server, and this package is available for free to all ODU students, staff, and faculty.[1] For those who have installed the CygWin package, a port of the XFree server is available.

[1] See the CS 252 "Library" page for further information.

Starting StarNet X-Win32

From the "Start" button menu, find and run `X-Win32`. Depending upon what version is installed, you will either see a new task appear on the task bar or a new icon in the task bar's icon tray, labelled with a blue "X". Right-click on this to bring up the menu. Look under the "Sessions" heading to see if someone has previously created a session definition for connecting to ODU CS machines: likely targets would be fast.cs.odu.edu, lab.cs.odu.edu, or dilbert.cs.odu.edu.

IF no such session has been defined, you will have to make your own. Start the `X-Util32` program, and select "New session". Fill in the requested information to create an `rexec` session running the command

```
/usr/openwin/bin/xterm -ls -display $DISPLAY
```

on `fast.cs.odu.edu` or one of the other CS machines (see this list). Leave the login name and password blank. You will be prompted for these each time you run the session (and if this is a public machine, you don't want it to save that kind of information where others might get to it!)

Run this session (e.g., Right-click on the X-Win32 icon to bring up the menu, and select your new session) to connect to the CS network. With luck, an `xterm`[2] window should appear. If nothing happens, return to the `X-Win32` menu, make sure that "Show Messages" is checked, and try again. This may give you some clue as what has gone wrong.

If you are unable to obtain an `xterm` window, another alternative is to first connect via telnet, set your DISPLAY variable to point to the machine you are sitting at and then launch `xterm`:

[2] `xterm` is a command window that allows you to enter commands to the remote Unix system, rather like telnet.

```
setenv DISPLAY local-machine-name:0
xterm
```

after which you can close the telnet window, if you desire.

Starting Exceed

To do a one-time launch, find and run Exceed's `XStart` from the Start button. You should receive a form similar to Figure 4.4

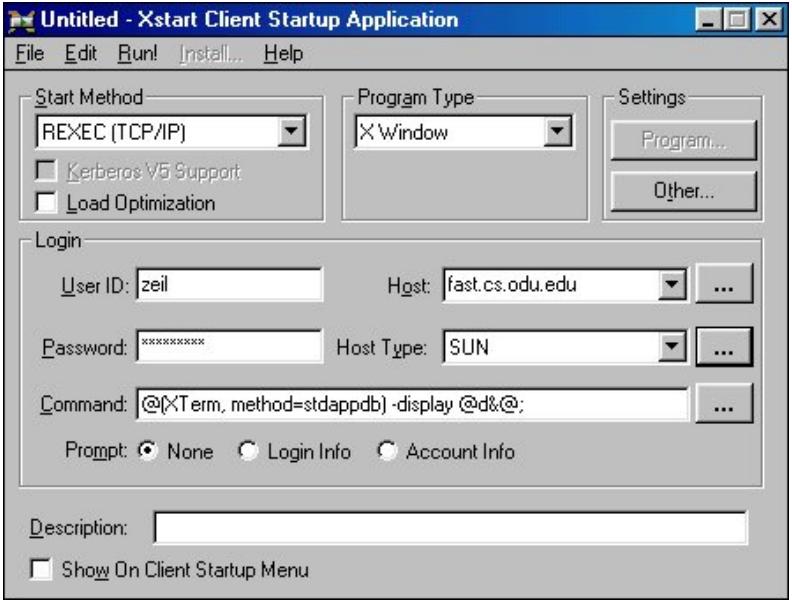
Alternatively, you can use Exceed's `Client Wizard` to create a session description that you can later click on to repeatedly launch xterms with the same connection information.

Either way, the information you need to supply is the same. Select either fast.cs.odu.edu or lab.cs.odu.edu as the host. Each of these are "SUN" for the host type. Select the rexec start method and the xterm application. The detailed command information should be OK as is[3] in the Wizard. For XStart, you will need to enter the command as shown in Figure 4.4. You can enter your user id, but if you are doing the Wizard on a public machine, don't enter your password.

XFree

I only recommend trying XFree if you have already installed CygWin and have become moderately familiar with it, are adept at downloading, and if you have a good tolerance for working out minor

[3]But if the connection fails, you may want to try changing the display parameter from "@d&@;" to an explicit display, as described earlier.



(full size)

Figure 4.4: Launching an Exceed Session

problems with software installations on your own. In fact, I would recommend that most people try StarNet's demo first. Then if you want to try XFree, you have a basis for comparison – you may well decide that you prefer the polish of the commercial version even if it does cost a few $$.

With that said, the installation instructions given in the user's manual at http://sources.redhat.com/cygwin/xfree/ are pretty straightforward. The default window manager supplied with CygWin's XFree port is `twm`, which is not my favorite, but is fairly simple to get going and is also discussed in some detail in your textbook.

At the end of the installation procedure, you should have a script named `startx`. (You will probably need to edit this script to, at the very least, adjust the parameters in the final line that describe your display size). From within a CygWin `bash` window, launch this script:

```
startx &
```

You should, shortly, have an X desktop before you with an `xterm` window inside.

Now, that xterm will be running on your local PC. To connect to an ODU machine, give the following commands from within that xterm window:

```
xhost +
rlogin fast.cs.odu.edu -l loginName
```

replacing `loginName` with your own Unix login. (If you prefer to connect to a specific ODU machine rather than the all-purpose "`fast`", you can also be more specific with your `xhost` command:

```
xhost +dilbert.cs.odu.edu
rlogin dilbert.cs.odu.edu -l loginName
```

Once you have logged in to the ODU CS machine, you will need to set your DISPLAY variable:

```
setenv DISPLAY localMachineAddress:0
```

4.1.3. Working in X

Once you have X working, try opening up some additional windows with commands like:

```
xterm &
xterm -bg Yellow -fg Green -sl 1000 -sb -geometry 40x16 &
xclock &
emacs &
xv /home/zeil/public_html/zeil.jpeg &
```

Exactly what you will see depends in part on the window manager. `X-Win32` uses MS Windows itself as its X window manager. XFree uses whatever manager you launch from within your `xstart` script.

A few things to keep in mind:

- Some window managers will not immediately open new windows, but will present you with an outline of the window first that you must move, with the mouse to the place on the screen you want, then click to lay the actual window down at that position.
- Any time you enter commands in Unix, you can place an ampersand (“&”) at the end of the command to run that command in the background. This “disconnects” the command from your key-

board (in that window). You get a prompt immediately and can enter your next command even if the one you just launched has not yet completed.

Now this capability is not all that useful if you're not running X. After all, if the program you are running needs input from you, it has been disconnected and can't see your subsequent keystrokes. Also, if that program produces output, it will still appear, but will be intermingled with the outputs of any new commands you have entered in the meantime. So, if you're not in X, the & is useful only for commands and programs that need no additional inputs and produce no additional outputs.

Under X, however, many useful programs open their own windows and direct their inputs and outputs through those new windows. For example, you would enter "`emacs &`" rather than "`emacs`", and "`netscape &`" rather than "`netscape`". Without the &, the window where you entered the command to launch a program would be useless to you until that program has finished. With the &, that program runs in its own window and the old window gets a new prompt and can still be used to issue more commands.

- Most programs that run under `X` support a very simple "cut-and-paste" facility. Simply drag the mouse across a block of text in any window while holding down the left mouse button. Then position the mouse into a window where you would like that text to be "typed". Click the middle mouse button, and the selected text will be sent to that window just as if you had typed it yourself.
- When `emacs` is run under `X`, this cut-and-paste feature is supported, but in a different fashion. Text that has been selected in another window by dragging the mouse can be retrieved in `emacs` by the command C-Y (^Y). Text that has been "killed" in `emacs` by C-K, C-W, or M-W can be inserted into other windows by clicking the middle mouse button.

4.1.4. Troubleshooting

"Cannot Open Display..."

This message is the most common problem that arises when trying to run X applications. Possible causes:

- You aren't running an X server — See subsection 4.1.2
- You forgot to set the DISPLAY variable or to specify it as a parameter when starting your X session — See subsection 4.1.2
- You misspelled your machine name or IP address when typing your DISPLAY value — try again
- You are running Starnet's X-Win32 on a PC within the CS Dept labs (on the second floor of the Education Building) — Try changing "`$DISPLAY`" in the command field of the session setup to "`$MYIP:0`":

  ```
  /usr/openwin/bin/xterm -ls -display $MYIP:0
  ```

  This may also be useful in other places when running X-Win32.
- You used a DISPLAY of the form `machineName:0` but used an abbreviated machine name, e.g., "daffodil" instead of "daffodil.cs.odu.edu". Try using the full name instead.
- You used a DISPLAY of the form `machineName:0` but for some reason your name isn't known to the client machine (your internet service provider might not be publishing your machine name to the general internet DNS (domain name service) — Try using your IP address instead.

- Your IP address begins with 10, 192, or 127 – You're working from behind a firewall or NAT server. See below.

Firewalls, NAT, & Internet Connection Sharing

One serious barrier to using X on some machines is the action of other programs that deliberately block signals from the X client from reachng the local server software.

Machine-to-machine communication over the internet is generally carried out by sending signals to one of several sockets, a software analogue of the hardware device that allows you to "plug in" other devices.

Sockets are identified by numbers. Some numbers are reserved for common internet services (telnet, ftp, email, and http all have their own specific socket number). X also communicates via sockets, using socket number $6000 +$ `display-value`. For example, if your DISPLAY is set this way:

```
setenv DISPLAY localMachineAddress:0
```

then X uses socket 6000, but if your display is set this way:

```
setenv DISPLAY localMachineAddress:2
```

then X uses socket 6002.

Firewalls are software security programs that many corporations use to protect their networks from outside hacking. One of the features of most firewalls is to block incoming communications to sockets other than the reserved socket numbers for email, http, or other services the corporation wants to support.

If you are trying to use X from behind a firewall, you may need to ask the system's administrator to permit connections to socket 6000. Another alternative is to try to connect via a Secure Shell (ssh), as described in section 15.

A similar problem arises with Network Address Translation (NAT), a scheme for allowing multiple computers on a local network to share a single internet connection. Microsoft Windows 98 and later offers a form of NAT they call "Internet Connection Sharing". Under this arrangement, for example, I have three computers in my home that can all access the internet through a single cable modem connection. To the outside world, I appear to have only a single computer there, because all three are sharing the same IP address.

NAT can confound X because when an X client tries to send information to your X server software on socket 600x, the NAT software has no clue as to which of its local machines it should route that information to. Luckily, some NAT packages allow you to configure the software to route specific socket requests to specific machines. Using these instructions, for example, I set up Windows Internet Connection Sharing to route socket 6000 to one machine and 6001 to another. Then I can run X server software on either machine and, by setting my DISPLAY appropriately, have the X windows appear on the machine I'm seated at.[4]

Again, another alternative is to try to connect via a Secure Shell (ssh), as described in section 15.

[4] And I know I've gotten it wrong when someone screams from the other room, "Dad! What's this emacs thing popping up on my screen?"

Running X via ssh

When you use ordinary telnet, ftp, or X over the internet, information to and from your PC are being sent as clear text. In theory, anyone else connected to the same local network as you (or as the machine with which you are communicating) could be reading the packets of information flowing by, writing down your passwords, etc.

Secure Shell (ssh) is an alternative set of internet tools that provide telnet-like and ftp-like capabilities but transmit all information in encrypted form, thus foiling potential eavesdroppers.

I mention htis here because ssh has another benefit – it can be used to "tunnel" the X protocol through many firewalls and NAT systems. To do this, you need two things:

1. The machine that will be running the X applications (the X client) must also be running an ssh server. Our lab.cs.odu.edu and fast.cs.oddu.edu machines all run ssh servers.

2. The machine where the X windows graphics are to be displayed (the X server) must have an ssh client program available.

(Notice that the X server goes with the ssh client and the X client goes with the ssh server. It's that reversal of client/server status that makes this work through many firewalls. Firewalls are often configured to let the protected machines serve as clients for any service they want to request, but not to allow anyone from the outside world to request a service from the protected machines. By reversing the client/server status, ssh makes X look to the firewall like a protected machine requesting service from the outside world.)

Where do you get an ssh client? Newer versions of X-Win32 and Exceed already include ssh support. ssh is also available as part of the CygWin package. Finally, check out this list from the OpenSSH project

at http://www.openssh.org.

To launch X via ssh:

X-Win32 and Exceed with built-in ssh support You simply need to select ssh when you create a new session.

CygWin Launch your X-server software (X-Win32, Exceed, or CygWin's XFree86) but don't try to start a session yet.

In any bash window:

```
export DISPLAY=localhost:0
ssh -X lab.cs.odu.edu /usr/openwin/bin/xterm
```

(You can substitute other names for the actual machine to connect with.)

Other ssh Clients Launch your X-server software (X-Win32, Exceed, or CygWin's XFree86) but don't try to start a session yet.

Invoke the ssh client program. Check the options/settings/preferences menu for something called "X forwarding", "X tunnelling", or the like. Make sure it is turned on. Then connect to the machine of your choice. You should get a telnet/xterm window on that machine from which you can launch X applications.

ssh: Some X applications work, Others cannot open display

There appears to have been a bug in one or more of the X libraries on the CS Dept's Unix (Sun Solaris) workstations. The net effect of this bug is that X applications compiled on certain machines in certain date ranges don't tunnel properly when invoked via ssh.

At the time I write this, xterm seems to work all the time. emacs seems to work all the time. ddd and xfig work on the lab.cs.odu.edu machines, but not on the fast.cs.odu.edu machines.

I recommend that if you are going to connect to X via ssh, you connect to lab.cs.odu.edu. If xterm works for you but other applications fail, try switching to fast.cs.odu.edu (although, at the moment, I am unaware of any applications that work on fast but fail on lab).

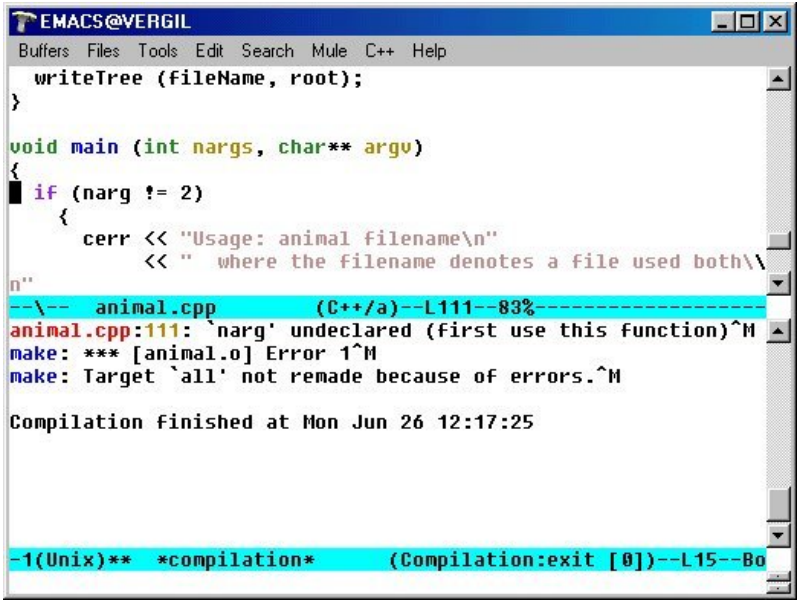
4.2. Editing under X

Editing is one task that can benefit tremendously from the availability of a windowing system, enhancing the ability to work with more than one document at a time, adding support for menus, mouse-clicks to reposition the cursor, scroll bars, etc.

If you have not already done so, try launching `emacs` from within an `xterm`. Notice that it pops up in a separate window. Compare this look (Figure 4.5) to `emacs` when run under telnet (Figure 4.6). Note that you now have functioning menus and scroll bars. You may see new use of color. A little experimentation will show that you can, indeed, reposition the cursor with a simple mouse click, and you may find that your `PageUp`, `PageDown` and other special keys now behave in an intuitive manner.

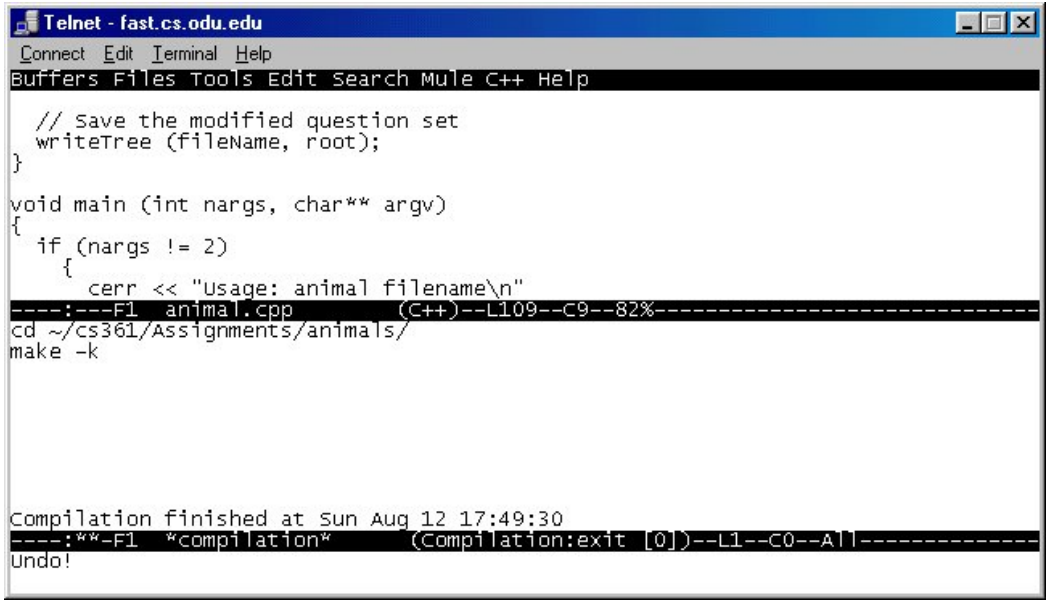
Many people who dislke `emacs` complain about having to learn a large number of keyboard command sequences to get things done, like moving the cursor, scrolling the text, etc. What you should be able to see, now, is that such key sequences are only necessary for relatively uncommon actions, or if you are not running in a windowed environment. After all, if you are running in telnet, how can you expect a mouse click to do anything? telnet is an exclusively text-oriented connection. In fact, `emacs` senses whether or not you are running in a windowing system, and takes advantage of it if you are, but leaves you with the key sequences as a backup mode when working under more primitive conditions.

`emacs` is certainly not the only available editor under X. You might want to also check out `xedit`, a simple text editor available on most Unix systems.


EMACS@VERGIL
Buffers Files Tools Edit Search Mule C++ Help
writeTree (fileName, root);
}
void main (int nargs, char** argv)
{
if (narg != 2)
{
cerr << "Usage: animal filename\n"
<< " where the filename denotes a file used both\\
n"
--\-- animal.cpp (C++/a)--L111--83%
animal.cpp:111: `narg' undeclared (first use this function)^M
make: *** [animal.o] Error 1^M
make: Target `all' not remade because of errors.^M
Compilation finished at Mon Jun 26 12:17:25
-1(Unix)** *compilation* (Compilation:exit [0])--L15--Bo

(full size)

Figure 4.5: emacs running under X



(full size)

Figure 4.6: emacs running under telnet

Chapter 5

EMail

5.1. Using Electronic Mail

Electronic mail, or "e-mail", for short, is an important part of the ODU CS environment. Besides being a useful way to exchange personal messages, e-mail is used by the Department for official announcements. Many instructors distribute grades by e-mail. They may send hints and corrections for projects and assignments that way, or distribute special files needed by all students in the class. E-mail may be the best way to pose a short question to your instructor outside of class, since you don't actually need to catch your instructor in person at a time when he/she's not busy with someone else.

Of course, you may already have an e-mail account at work, with the University, or with your own Internet Service Provider (ISP). If you prefer to receive all your mail at another account, you can set a forwarding address for your email.

5.1.1. E-Mail addresses

Just as with physical mail, you can't send someone e-mail unless you know their name and address. For e-mail, the name and address are usually combined as

name@machine

where *name* is the login name of the recipient and *machine* is the full name of the computer that processes their mail. This combination is generally called the person's "e-mail address".

For example, my login name is "zeil", and my mail is handled by the machine "cs.odu.edu", so my e-mail address is `zeil@cs.odu.edu`.

Actually all e-mail for CS Dept. login accounts is handled by `cs.odu.edu`. When you are sending mail to a user with the same mail handling machine, you can omit the "@" and everything that follows it. So if you are logged in to a CS Dept machine and want to send me e-mail you could just send it to `zeil`. But if you are logged in to a Teletechnet PC or a machine elsewhere on the Internet, you would need to give the full form, `zeil@cs.odu.edu`.

5.1.2. E-Mail Programs

There are several programs that you can use to get e-mail, and people tend to become rather fanatical about their personal favorite. The most basic of these is the Unix `mail` command, which also has the advantage of being universally available on any Unix machine. But `mail` is showing its age. New standards (the Multipurpose Internet Mail Extensions or MIME for short) have evolved to allow people to package files, graphics, sounds, etc., as part of an "extended" e-mail message. The `mail` command predates these standards and so cannot handle MIME e-mail. Also, `mail` is not the easiest e-mail program to learn.

I recommend the `pine` program for e-mail on our system. It is menu-driven, includes a substantial built-in help system, and can process and send MIME mail. I do occasionally fall back on the basic `mail` command, so I describe both of these in the following sections.

Later, you may want to check out the X-Windows `mailtool`, the `mush`, or `elm` programs, or the `vm` command for reading e-mail from within the `emacs` editor.

5.1.3. The Unix mail command

Sending

To send mail to someone with e-mail address *addr*, give the command

```
mail  addr
```

For example, you could send me mail via the command

```
mail zeil@cs.odu.edu
```

Although, if you are logged in to a CS Dept machine and want to send me e-mail you could just say

```
mail zeil
```

After you have given the `mail` command, you will be prompted for a subject line to indicate what your message is about. After that, you begin typing your message on the next line. When you are done, type ^D on a line by itself. You will then be prompted with "`Cc:`", which allows you to add the login names of other people to whom you would like to send a copy of your message. (Many people like to make a habit of cc'ing a copy to themselves.) If you do not want to send any extra copies, just hit the "Return" key. Your message will then be sent.

As you type your message, you can send special instructions to the mail program by entering any of the following at the start of a line:

~e Enter the editor named by the EDITOR environment variable. This is a good way to correct mistakes made on previous lines.

~r *filename* Insert the contents of a file into your mail message.

~p Print the message as it appears so far.

~m # If you are actually replying to a mail message that you received (see Receiving, below), this inserts the text of mail message number # into your reply.

Receiving

When you first log in, you will be informed if you have received e-mail. At that time, or anytime thereafter, you can use the `frm` command to list the messages awaiting.

To actually read your mail, give the command `mail` with no arguments. You should see a numbered list of your messages. If not, the command "`h`" (for **h**eaders) will list them. You can then read a message by typing it's number.[1]

After reading the message, you can take any of several actions:

r Send a **r**eply to the author of the message you just read.

R Send a **r**eply to the author of the message you just read and to anyone in the Cc: list of that first message.

dp Delete this message and move on to the next (if any).

[1] If you have no messages at the moment but would like to practice reading mail, try following the earlier instructions to send yourself a couple of messages. Then just wait a few minutes until `frm` indicates that your messages have arrived.

n Move on to the next message.

s *filename* Save a copy of this message in the specified file. If the file already exists, the message is added to the end.

5.2. Forwarding Addresses

If you prefer to read your e-mail on a different system, you can easily tell the Unix mailing system to forward all mail sent to your Unix e-mail address to a different address.

You will want to create, in your home directory (i.e., `cd ~`) a file named "`.forward`". The contents of this file should be a single line of text containing your preferred e-mail address.

You can create this file using your favorite text editor, or you can simply use the Unix `echo` command to write the desired text into the file. For example, if you wanted all your e-mail to be sent to `bogus@megacorp.com`, you would do the following

```
cd ~
echo "bogus@megacorp.com" > .forward
cat .forward
```

The final `cat` command should show the contents of the `.forward` file to be your desired address. (Note: Unix files that start with a "." are invisible to the normal `ls` command. To see them in a directory listing, you have to add the `-a` option: `ls -a`.)

Now, test it out! If you have a bad e-mail address in your `.forward` file, you could lose messages. So send yourself mail (to your ODU CS account). It should appear, in due course, at your preferred e-mail address. How long it actually takes depends on many factors. It may take only a few minutes. If after a few hours, you have not received the e-mail, delete your `.forward` file. Try again, if you wish. Or you might try on another day just in case the CS Dept. mail server, or the one at your preferred site, was temporarily out of commission. If you have repeated problems getting mail forwarded

quickly, you might want to rethink your desire to use this feature. For most people, this procedure works without much trouble.

You can actually have more than one forwarding address. The `.forward` file can contain a comma-separated list of forwarding addresses. For example, you might use

```
bogus@megacorp.com, bogus@home.net
```

Some people like to keep a backup copy of their mail on the CS Dept system, but to get their ”normal” mail somewhere else (e.g., because their mail server at work is crash-prone). This is possible, by making your e-mail address on the CS Dept. system one of the forwarding addresses, so that you forward a copy right back to yourself:

```
\yourLoginName, bogus@megacorp.com
```

The backslash is required: it helps prevent the mailer from consulting your `.forward` file a second time (which would lead to an infinite cycle of mail forwarding).

```
PINE 3.90   MAIN MENU                          Folder: INBOX  22 Messages

      ?     HELP              -  Get help using Pine
      C     COMPOSE MESSAGE   -  Compose and send/post a message
      I     FOLDER INDEX      -  View messages in current folder
      L     FOLDER LIST       -  Select a folder OR news group to view
      A     ADDRESS BOOK      -  Update address book
      S     SETUP             -  Configure or update Pine
      Q     QUIT              -  Exit the Pine program
```

Figure 5.1: Pine Main Menu

5.3. The PINE E-mail program

`pine` is a popular program for e-mail on our system. It is menu-driven, includes a substantial built-in help system, and can process and send MIME mail. It can be used over a text-based (e.g., telnet) connection with no loss of features.

To run `pine`, make sure that you have correctly identified your terminal kind. Then give the command

```
pine
```

You should see a menu resembling Figure 5.1. If you get a garbled screen instead, you probably have not set your terminal kind correctly.

```
To      :
Cc      :
Attchmnt:
Subject :
----- Message Text -----




^G Get Help  ^X Send      ^R Rich Hdr  ^Y PrvPg/Top ^K Cut Line  ^O Postpone
^C Cancel    ^D Del Char  ^J Attach    ^V NxtPg/End ^U UnDel Line^T To AddrBk
```

Figure 5.2: Sending mail with Pine

The most important choices are “C” to compose a message and send it to someone, and “I” to view an index of messages sent to you.

Sending Messages Type “C” to compose a message to send to someone. You should see a screen resembling Figure 5.2. One thing to note is the list of possible commands in the lower two lines of the screen. In almost any context, `pine` will list the commands available to you, including a command to get “help” information.

Use your up/down arrow keys to select the “To:” line at the top of the screen. Here you can enter the e-mail address you want to send a message to. Just below that, on the “CC:” line, you can add the e-mail addresses of any other people to whom you would like copies of your message sent. Two lines down is the “Subject:” line. Although you are not required to put anything here, proper e-mail “etiquette” calls

for all messages to carry a useful entry in the “Subject:” line.

Finally, move the cursor below the “Message Text” line, and you can begin typing your message.

When you are done and are ready to send your message, type ^X to send it.

A common variation on this procedure is when you need to send someone a copy of a file as part of your message. For example, if you are sending your instructor a question about some code you are writing, you might want to include the code in question as part of the message. Pine provides two ways to do this. The easiest is to go up to the “Attchmnt:” line near the top of the screen. Any file names you type on this line will be “attached” to the final message when it is sent (i.e., a copy is sent — your original files will be untouched). Alternatively, while you are typing your message you can ^R to insert a file directly into the text of your message.

Some points to keep in mind when deciding which approach to use are:

Attachment:

- Recipient of message must be using `pine` or some other MIME-capable mail program.
- Can be used to send non-text files (e.g., programs, graphics, etc) as well as standard text.
- Can send multiple files without confusion. File names are preserved as part of the attachment.

^R

- Recipient of message can use any mail program.
- Can only be used to send text files.
- File names are lost. If you try to include more than one file in a message, the boundaries between the files are likely to be unclear to the reader.

```
  PINE 3.90   FOLDER INDEX                 Folder: INBOX  Message 5 of 5

+ A 1   Nov  6 JohnDoe@elsewhere. (34,483) Looking for volunteers
+   2   Nov  8 JohnDoe@elsewhere.  (2,472) Still need volunteers
    3   Nov 13 MaryJones@aol.com   (4,310) Conference on Programming
    4   Nov 20 Root                (1,432) Re: your account
    5   Nov 22 Professor Z           (747) Overdue Homework



? Help       M Main Menu  P PrevMsg     - PrevPage    D Delete      R Reply
O OTHER CMDS V [ViewMsg]  N NextMsg   Spc NextPage    U Undelete    F Forward
```

Figure 5.3: Reading mail with Pine

Receiving Messages From the main menu screen, type “I” to get a list of the messages in your system “mail-box”. It will look something like Figure 5.3. The plus signs in front of some messages indicate that you have already read them. The “A” in front of message 1 indicates that you have already sent an answer to that message.

Using your up/down arrow keys, you can select any of these messages and then type “V” to view the message.

While viewing messages, refer to the bottom two lines of your screen for the commands to page up/down through long messages, to compose and send replies to a message, to “forward” a copy of the message to someone else, or to return to the main menu (see Figure 5.1).

If the message you are viewing is a MIME style message with attached files, another “V” command will allow you to view these files (if they are text, graphics, etc) or to save them in a directory of your

choice. Also, the “E” (Export) command will allow you to save the text of the current message in a file even if it is not a MIME message.

Folders After you have read some messages and try to exit from `pine`, `pine` will ask if you wish it to move your read messages out of your system mail-box (called the “INBOX”) into a “read-messages folder”. A *folder* is simply a container that can hold mail messages. `Pine` treats your system mail-box as simply a special form of folder. To see a list of your folders and to select one in which you want to view messages, use the “L” command from the `pine` main menu.

Moving read messages out of your system mail-box into another folder is often a good idea. It eases strain on the system resource area used for incoming e-mail. It means that when you enter `pine` and immediately hit the “I” key, you see your new mail right way instead of having it mixed in with old stuff. Finally, you can organize your saved mail by creating your own folders to save messages in. For example, you might have a folder for each class you are taking, to keep e-mail about different classes in separate containers. You can create new folders from inside the “L” command. To move a mail message that you are viewing into a folder, use the “S” save command.

Chapter 6

File Transfer

6.1. File Transfer

If you prepare files on one machine but want to use them on another, you need some means of transferring them. For example, if you edit files on your home PC or on a PC at one of the Teletechnet sites, you will eventually need to get those files onto the CS Department network. On the other hand, you may want to take files your instructor has provided off of that network for use on your home PC.

6.1.1. Before You Start: Understanding Transfer Modes

A complicating factor in transferring files from one computer to another is that you must decide whether the files you want to transfer should be treated as "text" of as "binary". Files that contain simple text, including program source code, should be transferred in "text" mode. Compiled programs, compressed files (e.g., *.zip or *.Z files), and word processor files with embedded formatting codes are generally transferred in "binary" mode.

The reason for this binary/text confusion is that Unix, MSDOS, IBM, and other systems disagree on how to represent basic text. For example, the end of a line in a Unix text file is represented by a single character (the ^J or "line-feed" character) while MSDOS uses a pair of characters at the end of each line (a ^M or "return" character followed by a ^J). Other operating systems have their own peculiarities. Most file transfer programs will, when transferring text, try to convert the transferred file into the appropriate format for the destination machine. These conversions may involve changing, adding, or deleting characters. Of course, if the file being transferred were not text but a compiled program, any such changes to individual bytes would be disastrous. Consequently, you need to be aware

at all times whether the files you are working with are text or binary.

The easiest way to tell (though not foolproof) is to try listing the file on your screen using the Unix "cat" command or the MSDOS *type* command. If it looks OK, its probably text. If not, or if in doubt, transfer it in binary mode.

If you do a transfer in binary mode and then discover that you have a text file with the wrong line terminators, you can correct that on the Unix side, as described in subsection 6.1.4.

6.1.2. Transferring Files Across the ODU Local Network: Samba

If you are sitting at a Windows PC that is part of either the CS Dept's own local network (e.g., in the CS Dept. labs) or part of the ODU OCCS local network (most ODU computer labs on the Norfolk, Virginia Beach, and Peninsula Graduate Center campuses)[1], then you can access your Unix account directories directly from within Windows. This is because the CS Dept. Unix file servers run a service called "Samba", a program that mediates file access between Windows and Unix systems.

To use Samba:

1. You need to know what file server is responsible for your Unix account directory. To find this out, log in to the Unix system and give the command:

```
df ~
```

[1] I'm not sure about other ODU remote locations. If anyone is able to try this at other locations, let me know if it works there.

You may get several lines of output (including some that say “permission denied” - don’t worry about them). For example:

```
Filesystem    1k-blocks   Used Available Use% Mounted on
df: /.cache/c2/.cfs_mnt_points/userve:_opt_43: Permission denied
userve:/stdnt1/yourName  657862  52066  29218  65% /home/yourName
```

Look for one line (probably the last) that lists your home directory (in this example, `/home/yourName` on the far right of the line. At the beginning of that line, just before the colon, is the name of your file server, `userve` in this example.

2. Now, let’s try to connect to your directories. From the Windows “Start” button, select “Run…” and type

   ```
   \\userve\yourName
   ```

   (replacing “yourName” by your Unix login name and “userve” by your file server name as determined in the prior step). You may be prompted for a password, or a login name and password. Supply your CS Unix login/password and, if all is well, a Windows Explorer window should open displaying the contents of your Unix home directory. You can now manipulate files in this Window just as you would in any Windows directory/folder, but the changes are occurring in your Unix directory.

3. Now that we know that it works, we can make this whole process more convenient by mapping a Windows drive letter to your Unix account, giving you a “fake” disk drive that actually accesses

your Unix files. From inside any Windows Explorer or "My Computer" window, select "Tools" (or right-click on the "My Computer" icon on the desktop) and select "Map Network Drive…". Select an unused drive letter, and enter the same `\\userve\yourName` string as in the last step for the "Folder". Make sure the "Recoonect at logon" box is checked. Finally, if your login name for logging into the PC is different from your CS Unix login name, look for a "Connect using a different user name" link, click on that and supply your Unix login information. Click on OK/Finish and within a few seconds, you should have a new drive available that actually maps onto your Unix account.

Two things to keep in mind when using Samba to access your Unix files from Windows:

- Watch your file permissions! Samba tries to set reasonable permission levels to Unix-hosted files that you create or modify from a Windows program, but it may not always do what you wanted. Be prepared to `chmod` any files you transfer in this way to what you really wanted.
- Samba will do all the transfers in binary mode. If that's not what you want, you will need to repair your text files as described in subsection 6.1.4.

6.1.3. Transferring Files Across the Internet: ftp

FTP (File Transfer Protocol) is the mechanism for transferring files over the internet. Although most browsers provide some support for `FTP`, they usually only permit downloads (from the remote machine to your local PC) and usually only permit access to public repositories, not to password-protected accounts.

You can only use FTP to transfer files to or from a machine that has been set up as an FTP server. In the ODU CS Dept., we have one such machine: ftp.cs.odu.edu.

Anonymous versus Private FTP

When you connect to an FTP server, you will be prompted for a login name and password. Thus, in the "normal" mode of FTP, you must have an account on the server system.

But some servers also have been set up to provide files to the public at large. By convention, servers that allow this do so by recognizing a special login name, "anonymous", for which almost any password is accepted. Again, by convention, users who log in as "anonymous" are expected to supply their own email address as the password.[2]

On the CS Dept server, ftp.cs.odu.edu, you can log in as "anonymous" to gain access to the public area. You cannot, however, access your own files from there. Alternatively, you can log in with your own Unix account login name and password, in which case you will have access to your own files and directories on the Unix network, but cannot access the public area.

This helps explain why you need to learn to use FTP rather than relying on your web browser for file transfer. Web browsers, when directed to an "ftp://" URL, do an anonymous login. If you need access to your own files and directories, that's no help.

[2] Some FTP servers actually check to see if the password supplied for an anonymous login appears to be a legimate email address (e.g., that it contains an '@' character somewhere, with at least one '.' after that).

Transferring Files

To transfer files via FTP, you need to have an FTP client program on the machine at which you are working. Not surprisingly, the way you actually launch and run the client depends upon just which client you have. MS Windows comes with an FTP client, called ftp. Cygwin provides another client, with the same name, which works almost identically. These offer a text-based interface. However, there are a variety of FTP clients, commerical and freeware, that offer window-GUI interfaces.

PCs on the labs maintained by OCCS have actually removed the default Windows ftp client program from many machines, and have installed the Hummingbird ftp client, which offers a familiar drag-and-drop style interface. Access this program in the "Start->Programs listing as "Hummingbird Connectivity...->Host Explorer->Telnet".

GUI interfaces can simplify FTP use, but these interfaces often create problems by hiding the text/binary settings, leading to corrupt transfers.

FTP via a Text Interface To use the MS Windows ftp, click the "Start" button, select "Run", and for the program name type

```
ftp ftp.cs.odu.edu
```

(Assuming you want to connect to the CS Dept. server. If you have reason to access another FTP server, just replace the machine name accordingly.)

If you are running the CygWin ftp, just type the same command into your shell.

You will then be prompted for your login name and your password. Enter those as usual.

Your next command should be

```
hash
```

This simply increases the amount of feedback you get about the progress made during file transfers.

Before actually transferring files, you must decide whether to use binary or text file transfer. If you want binary transfers, give the command

```
binary
```

and if you want text transfers, give the command

```
ascii
```

You can switch back and forth between these modes as necessary if you are transferring multiple files, some text and some binary.

Now you can use the commands `cd`, `pwd`, and `ls` to navigate the Unix directory structure as if you were in the shell. usually, you will `cd` to the directory of the remote machine in which you wish to download or upload files, do an `ls` to see what's there, and then proceed to transfer the specific files.

You can also change the directory of the local machine in which you will be working by giving the `lcd` (local `cd`) command. Note that the directory you give to this command must make sense in terms of the local machine's directory structure. For example,

```
lcd c:$\$courses$\$cs252
```

for the MS Windows `ftp` client, but

```
lcd /cygdrive/c/courses/cs252
```

under CygWin.

To get a file from the CS Unix machine to your local machine, the command is

```
get  filename
```

To put a file form your local machine onto the CS Unix machine, the command is

```
put  filename
```

Neither the `get` nor `put` commands can include wildcards (`* ?`) in the filename, but by changing the commands to `mget` and `mput`, you are allowed to use wild cards.[3]

To end your `ftp` session, the command is

```
quit
```

Here, then, is an example of an ftp session in which three files were downloaded from `~zeil/data` on the Unix network into `c:\misc\temp` on the local PC:

```
Connected to ftp.cs.odu.edu
```

[3] `mget` and `mput` will ask you, for each file matching the pattern you give, whether or not you really want to transfer it. If you are really sure you want to transfer every file matching the pattern, you can use the `prompt` command to turn this behavior off.

```
220 ProFTPD 1.2.Opre10 Server (ODU CS FTP Server)
User (ftp.cs.odu.edu(none))): zeil
331 Password required for zeil.
Password:
230 User zeil logged in
ftp> hash
Hash mark printing On  ftp: (2048 bytes/hash mark) .
ftp> lcd c:\misc\temp
Local directory now C:\misc\temp
ftp> cd data
250 CWD command successful
ftp> ls
200 Port Command successful
Directory of /home/zeil/data
file1.txt
file2.dat
file3.txt
file4.dat
226 Transfer Complete
ftp> ascii
200 Type set to A.
ftp> get file1.txt
```

```
200 PORT command successful
150 Opening ASCII mode data connection for file1.txt (20101 bytes).
#########
226 Transfer complete
ftp: 20627 bytes received in 1.17 seconds
ftp> binary
200 Type set to I.
ftp: mget *.dat
200 Type set to I.
mget: file2.dat? y
200 PORT command successful
150 Opening BINARY mode data connection for file2.dat (1001 bytes).

226 Transfer complete
ftp: 1001 bytes received in .23 seconds
mget: file4.dat? y
200 PORT command successful
150 Opening BINARY mode data connection for file4.dat (2123 bytes).
#
226 Transfer complete
ftp: 2123 bytes received in .35 seconds
ftp> quit
```

(You may notice that, during the ASCII mode transfer, the number of bytes transferred was larger than the size of the file. That's because, in an ASCII transfer from a Unix machine to a Windows machine, each new-line character is replaced by a carriage-return character followed by a new-line.)

FTP via Exceed ???

6.1.4. Problems and Inconsistencies

If you don't know whether to use binary or text transfer mode, try binary first.

If, however, you have transferred files to a Unix system and discover them to be full of ^M characters (you can see this by viewing the file in `emacs`), this is a sign that you should have used text mode. You can still recover, however, by using the command `dos2unix`:

```
dos2unix  file$_1$  file$_2$
```

to produce a new file $file_2$ from $file_1$ by converting the line ends to the Unix format.

On the other hand, if you have transferred files from a Unix system and find that the received files appear to consist of a single, extremely long line, you can use the command `unix2dos`:

```
unix2dos  file$_1$  file$_2$
```

to get a new file $file_2$ with ^M^J line terminators that can be transferred to your non-Unix machine instead of the original $file_1$.

Finally please note that, although easily transferred files may allow you to do most of the work of a programming assignment on your home PC, do not fall into the trap of believing that you can simply transfer the source code of a programming assignment and submit it unchanged to your instructor for grading on the Unix system. Different compilers for the same language often allow a variety of non-standard language extensions (or because of bugs, fail to properly compile standard language constructs). Allow yourself ample time (at least a few days) to port your code from one compiler to another.

Chapter 7

Program Development

7.1. Compilers

7.1.1. Compiling C and C++ in the Shell

Now that you know how to create and edit files, you can generate new programs. The most commonly used languages in the CS Department at the moment are C++, C, and Java.

The most popular C++ and C compilers are `g++` and `gcc`.[1]

Compiling a Single Source Code File

The simplest case for each compiler involves compiling a single-file program (or a program in which all files are combined by `#include` statements). For example, use an editor to prepare the following files:

hello.cpp

```
#include <iostream.h>
int main ()
{
  cout << "Hello from C++ !" << endl;
  return 0;
}
```

[1] Actually, `gcc` and `g++` are the same compiler being invoked with slightly different options.

hello.c

```
#include <stdio.h>
int main ()
{
  printf ("Hello from C!\n");
  return 0;
}
```

To compile and run these, the commands are:

```
g++ -g hello.cpp
./a.out
gcc -g hello.c
./a.out
```

The compiler generates an executable program called `a.out`. If you don't like that name, you can use the `mv` command to rename it. Alternatively, use a `-o` option to specify the name you would like for the compiled program:

```
g++ -g -o hello1 hello.cpp
./hello1
gcc -g -o hello2 hello.c
./hello2
```

Compiling Multiple Source Code Files

As you progress in your study of programming, you will discover that programs that can (or should) be written in a single source code file become increasingly rare. Most C and C++ programs are constructed from a collection of header (`.h`) and implementation (`.c`, `.cpp`) files.

A `.h` file is intended to be `#include`d from many different `.cpp` files that make up a single program. In fact, the earliest stage of compilation, the preprocessor, actually replaces each `#include` by the full contents of the included file.

A `.cpp` file is intended to be compiled once for any given build of the program.

A typical program will consist of many `.cpp` files. (See Figure 7.1) Usually, each class or group of utility functions will have their definitions in a separate `.cpp` file that defines everything declared in the corresponding `.h` file. The `.h` file can then be `#include`d by many different parts of the program that use those classes or functions, and the `.cpp` file can be separately compiled once, then the resulting object code file is linked together with the object code from other `.cpp` files to form the complete program.

Splitting the program into pieces like this helps, among other things, divide the responsibility for who can change what and reduces the amount of compilation that must take place after a change to a function body.

When you have a program consisting of multiple files to be compiled separately, add a `-c` option to each compilation. This will cause the compiler to generate a `.o` object code file instead of an executable. Then invoke the compiler on all the `.o` files together without the `-c` to link them together and produce an executable:

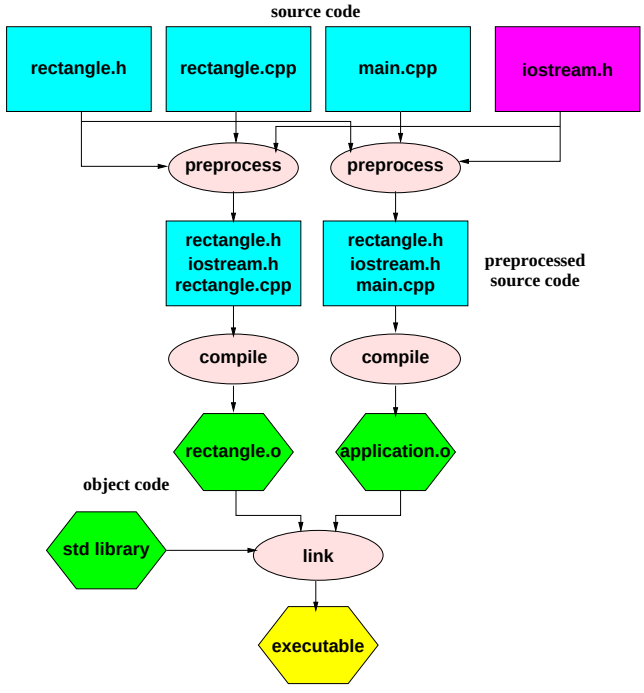


Figure 7.1: Building 1 program from many files

```
g++ -g -c file1.cpp
g++ -g -c file2.cpp
g++ -g -c file3.cpp
g++ -g -o programName file1.o file2.o file3.o
```

(If there are no other .o files in that directory, the last command can often be abbreviated to "`g++ -o programName -g *.o`".) The same procedure works for the `gcc` compiler as well.

Actually, you don't have to type separate compilation commands for each file. You can do the whole thing in one step:

```
g++ -g -o programName file1.cpp file2.cpp file3.cpp
```

But the step-by-step procedure is a good habit to get into. As you begin debugging your code, you are likely to make changes to only one file at a time. If, for example, you find and fix a bug in `file2.cpp`, you need to only recompile that file and relink:

```
g++ -g -c file2.cpp
g++ -g -o programName file1.o file2.o file3.o
```

An even better way to manage multiple source files is to use the `make` command.

Another useful option in these compilers is `-D`. If you add an option `-Dname=value`, then all occurrences of the identifier *name* in the program will be replaced by *value*. This can be useful as a way of customizing programs without editing them. If you use this option without a value, `-Dname`, then the compiler still notes that *name* has been "defined". This is useful in conjunction with compiler directive

`#ifdef`, which causes certain code to be compiled only if a particular name is defined. For example, many programmers will insert debugging output into their code this way:

```
⋮

x = f(x, y, z);
#ifdef DEBUG
   cerr << "the value of X is: " << x << endl;
#endif
y = g(z,x);
⋮
```

The output statement in this code will be ignored by the compiler unless the option `-DDEBUG` is included in the command line when the compiler is run.[2]

Sometimes your program may need functions from a previously-compiled library. For example, the `sqrt` and other mathmatical functions are kept in the "`m`" library (the filename is actually `libm.a`). To

[2] Zeil's 1st Rule of Debugging: Never remove debugging output. Just make it conditional. If you remove it, you're bound to want it again later.

Zeil's 2nd Rule of Debugging: Never leave your debugging code active when you submit your programs for grading. If the grader is using an automatic program to check the correctness of the output, unexpected output will make your program fail the tests. On the other hand, if the grader is reading the output to check its correctness, wading through extra output really ticks the grader off!

add functions from this library to your program, you would use the "`-lm`" option. (The "m" in "`-lm`" is the library name.) This is a linkage option, so it goes at the end of the command:

```
g++ -g -c file1.cpp
g++ -g -c file2.cpp
g++ -g -c file3.cpp
g++ -g -o programName file1.o file2.o file3.o -lm
```

The general form of gcc/g++ commands is

`g++` *compilation-options files linkage-options*

Here is a summary of the most commonly used options[3] for gcc/g++:

[3] A note for CygWin users: By default, CygWin gcc and g++ produce Windows "console" applications — applications designed to run from within a shell (`bash` or the Windows command line tool). They can, however, produce GUI applications with windows, menus, etc., by using `-l` to link in the windowing libraries. The list of libraries involved is rather long, so a shortcut option is provided: `-mwindows`.

Programming windowing code is a fairly involved process. I suggest getting a library that simplifies this process for beginners. The V library is a good choice, and has the additional advantage that code written for use with V can be compiled to produce either Microsoft Windows or Unix X windows programs.

Compilation Flags	
-c	compile only, do not link
-o *filename*	Use *filename* as the name of the compiled program
-D*symbol=value*	Define *symbol* during compilation.
-g	Include debugging information in compiled code (required if you want to be able to run the gdb debugger.
-O	Optimize the compiled code (produces smaller, faster programs but takes longer to compile)
-I *directory*	Add *directory* to the list of places searched when a "system" include (`#include <...>`) is encountered.
Linkage Flags	
-L *directory*	Add *directory* to the list of places searched for precompiled libraries.
-l*libname*	Link with the precompiled library `lib`*libname*`.a`

7.1.2. Error Messages

Unfortunately, once you start writing your own code, you will almost certainly make some mistakes and get some error messages from the compiler.

This is likely to lead to two problems: reading the messages, and understanding the messages.

Capturing the Error Messages

Unless you are a far better programmer than I, you will not only get error messages, you will get so many that they overflow your telnet/xterm window.

How you handle this problem depends upon what command shell you are running, but there are two general approaches. You can use redirection and pipes to send the error messages somewhere more convenient, or you can use programs that try to capture all output from a running program (i.e., the compiler).

We've talked before about how many Unix commands are "filters", working from a single input stream and producing a single output stream. Actually, there are 3 standard streams in most operating systems: standard input, standard output, and standard error. These generally default to the keyboard for standard input and the screen for the other two, unless either the program or the person running the program redirects one or more of these streams to a file or pipes the stream to/from another program.

1. Pipes and redirection

   We introduced pipes and redirection earlier. The complicating factor here is that what you want to pipe or redirect is not the standard output stream, but the standard error stream. So, for example, doing something like

   ```
   g++ myprogram.cpp > compilation.log
   ```

   or

   ```
   g++ myprogram.cpp | more
   ```

won't work, because these commands are only redirecting the standard output stream. The error messages will continue to blow on by.

How you pipe or redirect the standard error stream depends on the shell you are running:

Unix, running C-shell or TC-shell The `>` and `|` symbols can be modified to affect the standard error stream by appending a '&' character. So these commands do work:

```
g++ myprogram.cpp >& compilation.log
g++ myprogram.cpp |& more
```

A useful program in this regard is `tee`, which copies its standard input both into the standard output and into a named file:

```
g++ myprogram.cpp |& tee compilation.log
```

Win98/98/NT, running bash The sequence "`2>&1`" in a command means "force the standard error to go wherever the standard output is going". So we can do any of the following:

```
g++ myprogram.cpp 2>&1 > compilation.log
g++ myprogram.cpp 2>&1 | more
```

and we can still use `tee`:

```
g++ myprogram.cpp 2>&1 | tee compilation.log
```

2. Capturing compiler output

There are other ways to capture the output of a compiler (or of any running program). You can run the compiler within the emacs editor, which then gives you a simple command to move from error message to error message, automatically bringing up the source code file on the line cited in the error message. This works with both the Unix and the Win95/98/NT versions of emacs, and is the technique I use myself. `vim`, the "vi improved" editor will do the same.

Finally, in Unix there is a command called "`script`" that causes all output to your screen to be captured in a file. Just say

```
script log.txt
```

and all output to your screen will be copied into log.txt until you say

```
exit
```

`script` output can be kind of ugly, because it includes all the control characters that you type or that your programs use to control formatting on the screen, but it's still useful.[4]

Understanding the Error Messages

Cascading One thing to keep in mind is that errors, especially errors in declarations, can cascade, with one "misunderstanding" by the compiler leading to a whole host of later messages. For example, if you meant to write

[4] In particular, if you ever want to capture both the output of some program AND the stuff you typed to produce that output (e.g., so you can send it in an e-mail to someone saying "what am I doing wrong here?"), then `script` is the way to go.

```
string s;
```

but instead wrote

```
strng s;
```

you will certainly get an error message for the unknown symbol `strng`. However, there's also the factor that the compiler really doesn't know what type `s` is supposed to be. Often the compiler will assume that any symbol of unknown type is supposed to be an `int`. So every time you subsequently use `s` in a "string-like" manner, e.g.,

```
s = s + "abcdef";
```

or

```
string t = s.substring(k, m);
```

the compiler will probably issue further complaints. Sometimes, therefore, it's best to stop after fixing a few declaration errors and recompile to see how many of the other messages need to be taken seriously.

Backtracking A compiler can only report where it detected a problem. Where you actually committed a mistake may be someplace entirely different.

The vast majority of error messages that C++ programmers will see are

- syntax errors (missing brackets, semi-colons, etc.)
- undeclared symbols

- undefined symbols
- type errors (usually “cannot find a matching function” complaints)
- const errors

Let’s look at these from the point of view of the compiler.

Syntax errors Assume that the compiler has read part, but not all, of your program. The part that has just been read contains a syntax error. For the sake of example, let’s say you wrote:

```
x = y + 2 * x // missing semi-colon
```

Now, when the compiler has read only the first line, it can’t tell that anything is wrong. That’s because it is still possible, as far as the compiler knows, that the next line of source code wil start with a “;” or some other valid expression. So the compiler will never complain about this line.

If the compiler reads another line, and discovers that you had written:

```
x = y + 2 * x // missing semi-colon
++i;
```

it still won’t conclude that there’s a missing semi-colon. For all it knows, the “real” mistake might be that you meant to type “+” instead of “++”.

Now, things can be much worse. Suppose that inside a file `foo.h` you write

```
class Foo {
  Foo();
  int f();
  // missing };
```

and inside another file, `bar.cpp`, you write

```
#include "foo.h"

int g() {...}

void h(Foo) {...}

int main() {...}
```

Where will the error be reported? Probably on the very last line of `bar.cpp`! Why? Because until then, it's still possible, as far as the compiler knows, for the missing "`};`" to come, in which case `g`, `h`, and `main` would just be additional member functions of the class `Foo`.

So, with syntax errors, you know only that the real mistake occurred on the line reported or earlier, possibly in an earlier-`#include`'d file.

undeclared and undefined symbols When you forget to declare or define a type, variable, funciton, or other symbol, the compiler doesn't discover that anything is wrong until you try to use the unknown symbol. That, of course, may be far removed from the pace where you should have declared it.

type errors When you use the wrong object in an expression or try to apply the wrong operator/function to an object, the compiler may detect this as a type mismatch between the function and the expression supplied as the parameter to that function. These messages seem to cause students the most grief, and yet the compiler is usually able to give very precise descriptions of what is going wrong. The line numbers are usually correct, and the compiler will often tell you exactly what is going wrong. That explanation, however, may be quite lengthy, for three reasons:

1. Type names, especially when templates are involved, can be very long and messy-looking.
2. Because C++ allows overloading, there may be many functions with the same name. The compiler will have to look at each of these to see if any one matches the parameter types you supplied. Some compilers report on each function tried, explaining why it didn't match the parameters in the faulty call.
3. If the function call was itself produced by a template instantiation or an inline function, then the problem is detected at the function call (often inside a C++ standard library routine) but the actual problem lies at the place where the template was used/instantiated. So most compilers will list both the line where the error was detected and all the lines where templates were instantiated that led to the creation of the faulty call.

So, to deal with these, look at the error message on the faulty function call. Note what function/operator name is being complained about. Then look at the line where the faulty call occurred. If it's inside a template or inline function that is not your own code, look back through the "instantiated from" or "called from" lines until you get back into your own code. That's probably where the problem lies.

Here's an example taken from a student's code:

```
g++ -g -MMD -c testapq.cpp
/usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.2/../../../../includ
stl_relops.h: In function 'bool operator ><_Rb_tree_iterator<pair<con
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> >(const _Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> &, const _Rb_tree_iterator<pair<const
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> &)':
adjpq.h:234:   instantiated from 'adjustable_priority_queue<
 PrioritizedNames,map<PrioritizedNames,int,CompareNames,allocator<int
 ComparePriorities>::percolateDown(unsigned int)'
adjpq.h:177:   instantiated from 'adjustable_priority_queue<Prioritiz
 map<PrioritizedNames,int,CompareNames,allocator<int> >,
 ComparePriorities>::makeHeap()'
adjpq.h:84:   instantiated from here
 /usr/local/lib/gcc-lib/sparc-sun-solaris2.7/2.95.2/../../../../inclu
 g++-3/stl_relops.h:43: no match for 'const _Rb_tree_iterator<pair<co
 PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
 PrioritizedNames,int> *> & < const _Rb_tree_iterator<pair<const
```

```
PrioritizedNames,int>,pair<const PrioritizedNames,int> &,pair<const
PrioritizedNames,int> *> &'
```

Now, that may look intimidating, but that's mainly because of the long type names (due to template use) and the long path names to files from the C++ standard library. Let's strip that down to the essentials:

```
g++ -g -MMD -c testapq.cpp
stl_relops.h: In function 'bool operator >:
adjpq.h:234:   instantiated from 'percolateDown(unsigned int)'
adjpq.h:177:   instantiated from 'makeHeap()'
adjpq.h:84:   instantiated from here
stl_relops.h:43 no match for ... < ...
```

(This one is actually worse than most error messages, becuase it's easy to miss the "`<`" operator amist all the `<...>` template markers.)

The problem is a "no match for" a less-than operator call in line 43 of a template within the standard library file `stl_relops.h`. But that template is instantiated from the student's own code (adjpq.h) and so the thing to do is to look at those three lines (234, 177, and 84) for a data type that is supposed to support a less-than operator, but doesn't.

const errors Technically, "const"-ness is part of a type, so while sometimes these get special messages of their own, often they masquerade as ordinary type errors and must be interpreted in the same way.

(Continued from here.)

7.1.3. Compiling Java in the Shell

Java programs get compiled into object code for an imaginary CPU called the "Java Virtual Machine" (JVM). Consequently, you can't execute compiled Java code directly. You must run a program that simulates a JVM and let that simulated computer execute the Java code.

That may seem a little convoluted, but the JVM simulator is easier to write than a "true" compiler. Consequently, JVM simulators can be built into other programs (such as web browsers), allowing Java code compiled on one machine to be executed on almost any other machine. By contrast, a true native-code compiler (e.g., g++) produces executables that can only be run on a single kind of computer.

The command to compile Java code is "`javac`" ("c" for compiler) and the command to execute compiled Java code is "`java`". So a typical sequence to compile and execute a single-file Java program would be

```
javac -g MyProgram.java
java MyProgram
```

Unlike most programming languages, Java includes some important restrictions on the file names used to store source code.

- Java source code is stored in files ending with the extension "`.java`".
- Each Java source code file must contain exactly one public class declaration.

- The base name of the file (the part before the extension) must be the same (including upper/lower case characters) as the name of the public class it contains.

So the command

```
javac -g MyProgram.java
```

compiles a file that must contain the code:

```
public
class MyProgram ...
```

The output of this compilation will be a file named `MyProgram.class` (and possibly some other `.class` files as well).

If we have a program that consists of multiple files, we can simply compile each file in turn:

```
javac -g MyProgram.java
javac -g MyADT.java
```

but this might not be necessary. If one Java file imports another, then the imported file will be automatically compiled if no `.class` file for it exists.

So, if the file `MyProgram.java` looked like this

```
import MyADT;

public
class MyProgram ...
```

then compiling `MyProgram.java` would also compile `MyADT.java`.[5]

Java programs come in two forms: applets and applications. Applets are placed on web servers and can be launched from HTML web pages so that the Java code runs whenever someone browses that page. Applications, on the other hand, are more like traditional programs that get invoked directly by the user.

To run a Java application, we use `java`:

```
java MyProgram
```

which looks for a file named `MyProgram.class`. Within that file, it looks for a compiled version of a function `Main` that must have been declared this way:

```
public static
void main (java.lang.String[] args) {
  ⋮
}
```

and executes that function.

As Java programs get larger, programmers usually begin to group their classes into packages. packages can also be grouped (nested) inside other packages. Again, Java has rules that cause the program's modular structure to be reflected in the file structure used to store it. If a class is declared to be inside

[5]Beware, however, after the first compilation. `javac` by default only checks to see if an appropriately named `.class` file exists. If you subsequently make changes to `MyADT.java` and then recompile `MyProgram.java`, the compiler will not realize that `MyADT.java` needs to be recompiled as well.

one or more levels of nsted packages, then each package name is used as a directory name when storing the source code file.

For example, if we had source code like this:

```
package Project.Utilities;

class MyADT { ⋮
```

and

```
package Project;;

import Project.Utilities.MyADT;

class MyProgram {
   ⋮
  public static
  void main (java.lang.String[] args) {
    ⋮
  }
}
```

then we would need a directory/file structure as shown in Figure 7.2 named “Project”. Inside that

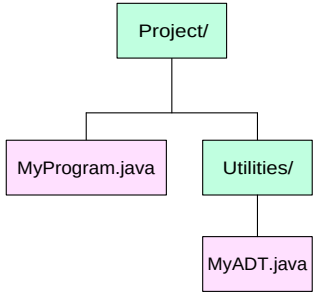


Figure 7.2: Sample Java Package Structure

directory would be the `MyProgram.java` file and another directory named “Utilities”, and the `MyADT.java` function goes inside that `Utilities` directory.

Now, here’s the part that trips up many a Java programmer:

- When you compile and execute code in Java packages, you must always do so from the directory at the top of the package structure.
- The `javac` compilation command takes the name of the source code file.
- The `java` execution command takes the name of the class contianing the`main` function.

So the compilation and execution commands would be

```
javac -g Project/Utilities/MyADT.java
```

```
javac -g Project/MyProgram.java
java Project.MyProgram
```

As with g++, there are several options that you may choose to employ when compiling and executing Java code. Here is a summary of the most commonly used ones:

Compilation Flags	
-classpath *pathlist*	Add the directories named in the pathlist (multiple directories may be separated by ':') to the list of places searched when `import`ing other Java source code.
-g	Include debugging information in compiled code (required if you want to be able to run the gdb debugger.
-depend	Check each imported class to see if its source code has been changed since it was last compiled. If so, automatically recompile it.
-deprecation	Check the code for features that used to be legal in Java, but are expected to become illegal in the near future.
-O	Optimize the compiled code (produces smaller, faster programs but takes longer to compile)
Execution Flags	
-classpath *pathlist*	Add the directories named in the pathlist (multiple directories may be separated by ':') to the list of places searched when `import`ing other compiled Java .class files.

7.2. Compiling in emacs

When your programs contain mistakes, compiling them in the shell can result in large numbers of error messages scrolling by faster than you can read them. For this reason, I find it best to compile from within the `emacs` editor.

Get into `emacs` and call up one of the "hello" programs. Change it so that it contains one or more syntax errors, and save this file. Now give the `emacs` command: `M-x compile`. At the bottom of the screen, you will be asked for the compile command. If the suggested command is not what you want (it won't be, until you learn to use `make` to manage your program compilations), then type in the proper command just as if you were typing it into the shell. `emacs` will invoke the compiler, showing its output in a window. Figure 7.3 shows a typical emacs session after such a compilation.

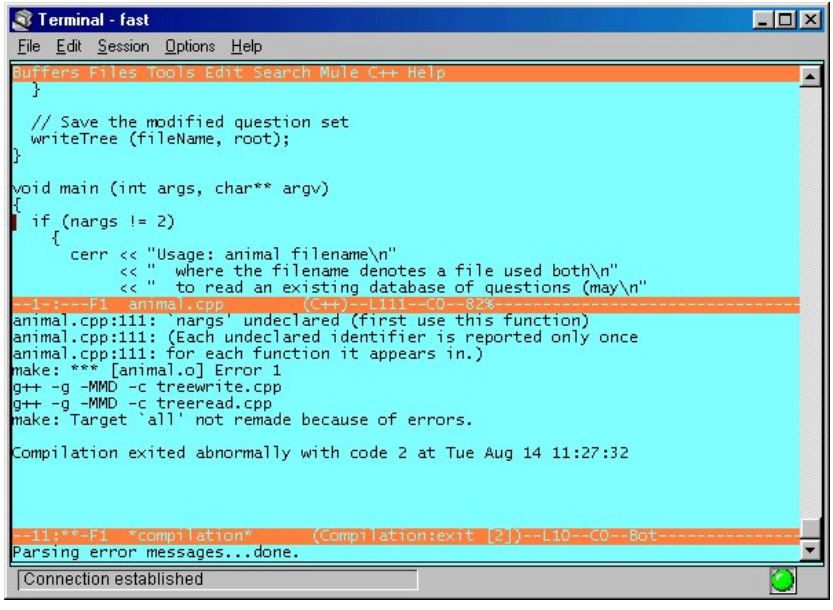
In this case, there should be one or more error messages. The `emacs` command `C-x ‘` will move you to the source code location of the first error. Each subsequent use of `C-x ‘` will move you to the next error location in turn, until all the reported error messages have been dealt with.[6]

7.2.1. Compiling with vim

`vim` ("vi improved") is an reasonable alternative to `emacs` on many Unix systems. It provides syntax-highlighting editor, can launch the compiler and will display the error messages, synchronized with the source code (Figure 7.4). It provides no interface to the debugger, however.

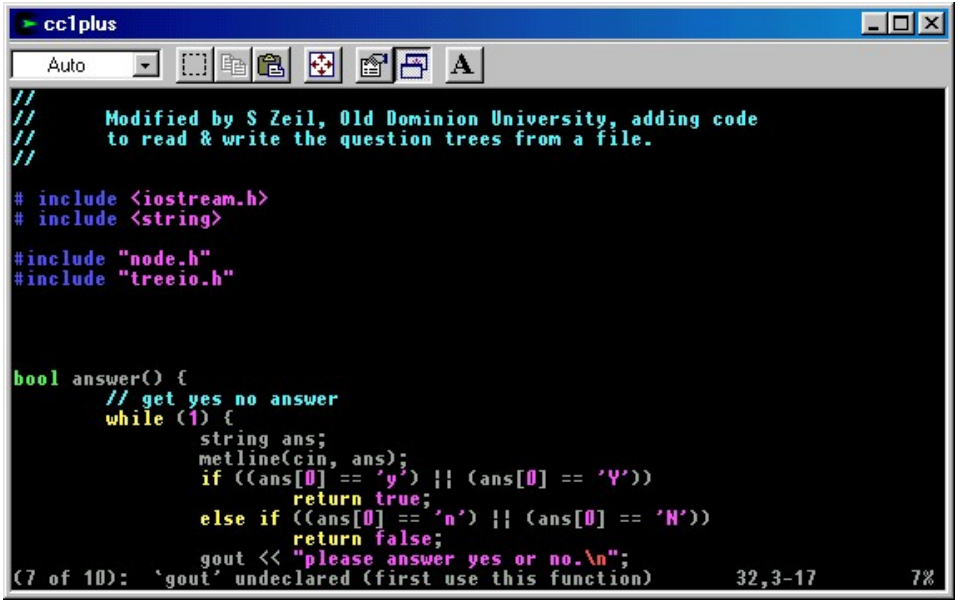
To learn the basics of running vim, give the command `vimtutor`.

[6] Note carefully that the second character in the `C-x ‘` command is the "backwards" apostrophe, not the regular one.


Terminal - fast
File Edit Session Options Help
Buffers Files Tools Edit Search Mule C++ Help
}
// Save the modified question set
writeTree (fileName, root);
}
void main (int args, char** argv)
{
if (nargs != 2)
{
cerr << "Usage: animal filename\n"
<< " where the filename denotes a file used both\n"
<< " to read an existing database of questions (may\n"
--1-:---F1 animal.cpp (C++)--L111--C0--82%
animal.cpp:111: `nargs' undeclared (first use this function)
animal.cpp:111: (Each undeclared identifier is reported only once
animal.cpp:111: for each function it appears in.)
make: *** [animal.o] Error 1
g++ -g -MMD -c treewrite.cpp
g++ -g -MMD -c treeread.cpp
make: Target `all' not remade because of errors.
Compilation exited abnormally with code 2 at Tue Aug 14 11:27:32
--11:**-F1 *compilation* (Compilation:exit [2])--L10--C0--Bot
Parsing error messages...done.
Connection established

(full size)

Figure 7.3: Compiling a program in emacs, with syntax errors



(full size)

Figure 7.4: The vim (vi improved) editor

After you worked through the basics, cd into a directory where you have a C++ program and a makefile, and load one of the source files (e.g., `vim foo.cpp`). The command "`:make`" (including the leading colon ":") will run the `make` program to attempt compilation. If you have errors, the command `:cc` will display the first one, `:cn` moves you to the next error, and `:cp` moves you to the previous one.

7.3. Project Management with Make

When you begin to develop projects that involve multiple files that need to be compiled or otherwise processed, keeping them all up-to-date can be a problem. Even more of a problem is passing them on to someone else (e.g., your instructor) and expecting them to know what to do to build your project from the source code.

The Unix program `make` is designed to simplify such project management. In a *makefile*, you record the steps necessary to build both the final file (e.g., your executable program) and each intermediate file (e.g., the `.o` files produced by compiling a single source code file).

We say that a file `file1` *depends upon* a second file `file2` if the `file2` is used as input to some command used to produce `file1`.

When the `make` program is run, it then checks to be sure that all of the needed files exist, and that each needed file has been updated more recently than all of the files it depends upon. The key bits of information in a makefile, therefore are

- For each file, a list of other files it depends upon, and
- The command used to produce the dependent file from the files it depends upon.

A makefile may also include various macros/abbreviations designed to simplify the task of dealing with many instances of the same commands or files.

Suppose that we are engaged in a project to produce 2 programs, `progA` and `progB`. `progA` is produced by compiling files `utilities.c`, `progA1.cpp`, and `progA2.cpp` and linking together the resulting `.o` files. Program `progB` is produced by compiling file `utilities.c` and `progB1.cpp`

and linking together the resulting `.o` files. All of the `.c` and `.cpp` files have `#include` statements for a file `utilities.h`. Also, both of the `.cpp` files have an `#include` statement for a file `progA1.h`.

Here is a makefile for this project. This file should reside in the project directory, and should be called "`Makefile`" or "`makefile`".

```
progA: utilities.o progA1.o progA2.o
        g++ -g -DDEBUG utilities.o progA1.o progA2.o
        mv a.out progA

progB: utilities.o progB1.o
        g++ -g -DDEBUG utilities.o progB1.o
        mv a.out progB

utilities.o: utilities.c utilities.h
        g++ -g -DDEBUG -c utilities.c

progA1.o: progA1.cpp utilities.h progA1.h
        g++ -g -DDEBUG -c progA1.cpp

progA2.o: progA2.cpp utilities.h progA1.h
        g++ -g -DDEBUG -c progA2.cpp

progB1.o: progB1.cpp
```

```
g++ -g -DDEBUG -c progB1.cpp
```

The key information in a makefile consists of a variety of rules for producing "target" files. Each target rule begins with a single line containing the name of the file to produce, a colon, and then a list of all files that serve as inputs to the commands that produce the file. Following that are any number of command lines that give the Unix commands to actually produce the file. Each command line starts with a "Tab" character (invisible in this listing).

Suppose that, with just this `Makefile` and the various source code files in your directory, you issued the command `make progB`. `make` reads the `Makefile` and notes that `progB` depends upon `utilities.o` and `progB1.o`. Since neither of these files exists, `make` sets out to create them. `utilities.o` depends upon `utilities.c` and `utilities.h`. Since these files exist and do not themselves depend upon anything else, `make` will issue the command to create `utilities.o` from them. This command is the "standard" command for making a `.o` file from a `.c` file:

```
gcc -g -DDEBUG -c utilities.c
```

Next `make` looks at `progB1.o`. It depends upon `progB1.cpp` which exists and does not depend upon anything else. So `make` uses the standard command for C++ files:

```
g++ -g -DDEBUG -c progB1.cpp
```

Now that both `.o` files have been created, `make` proceeds to build its main target, `progB`, using the command lines provided for that purpose:

```
g++ -g -DDEBUG utilities.o progB1.o
```

and the `progB` program has been created.

Now suppose that we immediately give the command "`make progA`" (or just "`make`", since by default `make` builds the first target when none is explicitly given). Then the following commands would be performed:

```
g++ -g -DDEBUG -c progA1.cpp
g++ -g -DDEBUG -c progA2.cpp
g++ -g -DDEBUG utilities.o progA1.o progA2.o
mv a.out progA
```

Note that `utilities.c` is not recompiled, because `make` would notice that `utilities.o` already exists and was created more recently than the last time when either `utilities.c` or `utilities.h` was changed.

If you want to test your makefile without actually performing the commands, add a `-n` option to your command (e.g., `make -n progB`) and `make` will simply list the commands it would issue without actually doing any of them.

7.3.1. Symbols

Thinking ahead, we might realize that we won't always want to compile with the flags "`-g -DDEBUG`" (the significance of which will be introduced in the debugging section).

We can make our makefile more flexible by gathering things that might need to be changed later into a symbol. `make` allows us to define symbols like this

```
SymbolName=string
```

and later use that symbol like this: `$(SymbolName)`.

So we could modify our makefile as follows:

```
# Macro definitions for "standard" language compilations
#
#  First, define special compilation flags. These may change when
#  we're done testing and debugging.
CPPFLAGS=-g -DDEBUG
#

#
# Targets:
#
progA: utilities.o progA1.o progA2.o
        g++ $(CPPFLAGS) utilities.o progA1.o progA2.o
        mv a.out progA

progB: utilities.o progB1.o
        g++ $(CPPFLAGS) utilities.o progB1.o
        mv a.out progB
```

```
utilities.o: utilities.c utilities.h
        g++ $(CPPFLAGS) -c utilities.c

progA1.o: progA1.cpp utilities.h progA1.h
        g++ $(CPPFLAGS) -c progA1.cpp

progA2.o: progA2.cpp utilities.h progA1.h
        g++ $(CPPFLAGS) -c progA2.cpp

progB1.o: progB1.cpp
        g++ $(CPPFLAGS) -c progB1.cpp
```

7.3.2. Default Rules

Makefiles can be simplified by introducing default rules for forming one kind of file from another. Here is an equivalent makefile that defines appropriate default rules:

```
# Macro definitions for "standard" language compilations
#
#  First, define special compilation flags. These may change when
#  we're done testing and debugging.
CPPFLAGS=-g -DDEBUG
#
```

```
#  The following is "boilerplate" to set up the standard compilation
#  commands:
.SUFFIXES:
.SUFFIXES: .cpp .c .cpp .h .o
.c.o: ; gcc $(CPPFLAGS) -c $*.c
.cpp.o: ; g++ $(CPPFLAGS) -c $*.cpp
#
# Targets:
#
progA: utilities.o progA1.o progA2.o
        g++ $(CPPFLAGS) utilities.o progA1.o progA2.o
        mv a.out progA

progB: utilities.o progB1.o
        g++ $(CPPFLAGS) utilities.o progB1.o
        mv a.out progB

utilities.o: utilities.c utilities.h

progA1.o: progA1.cpp utilities.h progA1.h

progA2.o: progA2.cpp utilities.h progA1.h
```

```
progB1.o: progB1.cpp
```

In the "SUFFIXES" area, standard commands are defined for producing a `.o` file from a `.c` or `.cpp` file. Of course these standard commands simply invoke the C or C++ compilers. Command lines are not needed if the standard commands from the "Suffixes" area can be used to build the desired file.

7.3.3. Common Conventions

So far, we have talked about using `make` exclusively with compilation. But a makefile can control almost any sequence of operations that build one kind of file out of others.

Certain conventions have arisen that you will find useful in designing your own makefiles and in using the makefiles of others. Most of these involve certain "artificial" targets that you can use when issuing the `make` command.

These are just conventions. They don't happen automatically, but most people who design makefiles set them up to work this way.

all The target `all` compiles and builds everything (except, sometimes, documentation). So one of the more common ways ot invoke make is to say

```
make all
```

The `all` target rule, if it is present, is always given as the first rule in the makefile. The `make` command, if not given any target, always goes to the first target rule. So, in most makefiles,

```
make
```

will also compile and build everything.

clean The command `make clean` is often used to clean up a directory, deleting all the temporary files produced by `make all`, leaving only the original files (e.g., the source code) from which everything else can be rebuilt later, if desired.

install In programs that must be "installed" by placing them in special directories, this target controls the commands necessary to do that installation.

A common sequence for building and installing new Unix software is therefore:

```
make
make install
make clean
```

test Less common, runs a test suite to see if the program built successfully.

doc (or `docs`) May be used to build program documentation, user manuals, etc.

We can bring our sample makefile into conformity with these conventions as follows:

```
# Macro definitions for "standard" language compilations
#
#  First, define special compilation flags. These may change when
```

```
#  we're done testing and debugging.
CPPFLAGS=-g -DDEBUG
#
#  The following is "boilerplate" to set up the standard compilation
#  commands:
.SUFFIXES:
.SUFFIXES: .cpp .c .cpp .h .o
.c.o: ; gcc $(CPPFLAGS) -c $*.c
.cpp.o: ; g++ $(CPPFLAGS) -c $*.cpp
#
# Targets:
#
all: progA progB

clean:
rm progA progB *.o

progA: utilities.o progA1.o progA2.o
        g++ $(CPPFLAGS) utilities.o progA1.o progA2.o
        mv a.out progA

progB: utilities.o progB1.o
```

```
        g++ $(CPPFLAGS) utilities.o progB1.o
        mv a.out progB

utilities.o: utilities.c utilities.h

progA1.o: progA1.cpp utilities.h progA1.h

progA2.o: progA2.cpp utilities.h progA1.h

progB1.o: progB1.cpp
```

7.3.4. Is It Worth It?

Now, creating a makefile may seem like a lot of trouble the first time that you want to compile your program. The payoff comes while you are testing and debugging, and find yourself making changes to two or three files and then needing to recompile. Which files do you really need to recompile? It can be hard to remember some times, and the errors resulting from an incorrect guess may be hard to understand. `make` eliminates this problem (as well as just being easier to type than a whole series of recompilation commands). (This is why, when you give the `M-x compile` command in `emacs`, the default compilation command is "make" rather than a direct use of any particular compiler.)

Most of the details of generating a makefile can be automated. The `gcc`/`g++` compiler, for example, will actually write out the makefile rules that would determine when a given `.c` or `.cpp` file needs

to be recompiled. I've used this idea in this "self-constructing" Makefile. To use it, copy it into your working directory where you keep the source code files for any single program. Your copy must be named "Makefile". Edit your copy of the file to supply the appropriate program name, list of source code files needed for that program, and to indicate whether the final step (linking) should be done with the `C` (`gcc`) or `C++` (`g++`) compiler.

Now you can compile your program by saying `make`, and clean up afterwards with `make clean`.

As you continue to work with your code, just remember to keep the `OBJS` list in the Makefile up to date.

7.4. Debugging

Your program is producing incorrect output, or perhaps has crashed with no output at all. How do you find out why?

There's no easy answer to that. Debugging is hard work, and is as much an art as an engineering process. Basically, though, debugging requires that we reason backwards from the symptom (the incorrect output or behavior) towards possible causes in our code. This may require a chain of several steps:

Example:

Hmmm. The program crashed near the end of this loop:

```
for (int i = 0; i < numElements; ++i)
{
  cout << myArray[i] << endl;
}
myArray[0] = myArray[1];
```

Each time around the loop, the code prints `myArray[i]`. *One possible reason for wuch a crash would be if* `i` *was outside the legal range of index values for that array. Now, what could cause that? Maybe the loop exit condition is incorrect. Maybe the array is too small. Maybe we counted the number of items in the array incorrectly.*

As we work backwards, we form hypotheses about what might be going wrong, for example, "`i` was outside the legal range of index values for that array".

An integral part of debugging is testing those hypotheses to see if we are on the right track. This often involves seeking additional information about the execution that was not evident from the original program output, such as how often the loop is executed, whether we actually exited the loop, what values of `i` or `numElements` were employed just before the crash.

An automated debugger can help us in these endeavors. Such a debugging tool typically allows us to set breakpoints at different locations in the code, so that if execution reaches one of those locations, the execution will pause. Debuggers also allow us to examine the program state, printing out the values of variables and the history of function calls that brought us to the current location. Debuggers typically let us step through the code, one statement at a time, so that we can watch the execution at a very fine level.

An automatic debugger is a powerful tool for aiding our reasoning process during debugging. It can be invaluable in those frustrating cases where the program crashes without any output at all, as the debugger will usually take us right to the location where the crash occurred.

Automatic debuggers can also be a tremendous waste of time, however. It's all too tempting to single-step through the code aimlessly, hoping to notice when something goes wrong. Debuggers are best used to augment the reasoning process, not as a substitute for it. In that vein, it's worth noting first the alternatives to automated debuggers (even if that takes us somewhat beyond the scope of a "Unix" course):

7.4.1. Debugging Output

One of the easiest ways to gather information about what's happening in a program is to add output statements to the code. These can show the values of key variables and also serve as documentation of

what statements in the code were actually reached during execution. In our example above, the easiest way to test our hypothesis about `i` going out of bounds would be to alter our code like this:

```
for (int i = 0; i < numElements; ++i)
{
  cerr << "i: " << i << endl;
  cout << myArray[i] << endl;
}
cerr << "Exited loop" << endl;
myArray[0] = myArray[1];
```

Note that we send the debugging information, not to the standard output, but to the standard error stream. This may or may not be significant, but in many programs standard output may be redirected to files or to other programs, and we would not want to introduce new complications by having this unanticipated extra output included in that redirection.

This solution is not perfect, however. For one thing, we need to remember that this extra output is not acceptable in the final version of the code and must be removed. Actually removing the debugging output statements may not be a good idea anyway. In my experience, I have often removed debugging output statements after fixing one bug, only to discover another bug and wish that I had the same output available once more. So, I seldom remove debugging output, preferring instead to comment it out:

```
for (int i = 0; i < numElements; ++i)
{
  // cerr << "i: " << i << endl;
```

```
  cout << myArray[i] << endl;
}
// cerr << "Exited loop" << endl;
myArray[0] = myArray[1];
```

But, if we wind up with lots of debugging output like this in our program, we may have to hunt to find and remove it all before turning in our final program. A better solution is to use conditional compilation:

```
for (int i = 0; i < numElements; ++i)
{
#ifdef DEBUG
  cerr << "i: " << i << endl;
#endif
  cout << myArray[i] << endl;
}
#ifdef DEBUG
cerr << "Exited loop" << endl;
#endif
myArray[0] = myArray[1];
```

Now our debugging output will only be compiled only if the compile-time symbol `DEBUG` is set. This can be done by defining it at the start of the file (or in a `.h` file that is `#include`'d by this one):

```
#define DEBUG 1
```

or by defining it when we compile the code:

```
g++ -g -c -DDEBUG myProgram.cpp
```

For the final program submission, we simply omit these definitions of `DEBUG`, thereby turning off all our debugging output at once.

Another possibility is to use the macro facilities of C and C++ to define special debugging commands that, again, are active only when `DEBUG` is defined: For example, I sometimes have a header file named `debug.h`:

```
#ifdef DEBUG

   #define dbgprint(x) cerr << #x << ": " << x << endl
   #define dbgmsg(message) cerr << message << endl

#else

   #define dbgprint(x)
   #define dbgmsg(message)

#endif
```

These are macros. The C/C++ compiler will now replace any strings of the form `dbgprint(...)` and `dbgmsg(...)` in your source code by the rest of the macro before actually compiling your code.[7]

So we can then write:

```
#include "debug.h"
   ⋮
for (int i = 0; i < numElements; ++i)
{
  dbgprint(i);
  cout << myArray[i] << endl;
}
dbgmsg("Exited loop");
myArray[0] = myArray[1];
```

If we compile that code this way:

```
g++ -g -c -DDEBUG myProgram.cpp
```

then what actually gets compiled is

```
#include "debug.h"
   ⋮
```

[7] The `#x` is a special trick supported by the C/C++ macro processor. It takes whatever string `x` stands for, and puts it inside quotes.

```
for (int i = 0; i < numElements; ++i)
{
  cerr << "i" << ": " << i << endl;
  cout << myArray[i] << endl;
}
cerr << "Exited loop" << endl;
myArray[0] = myArray[1];
```

but if we compile the code this way:

```
g++ -g -c myProgram.cpp
```

then what actually gets compiled is

```
#include "debug.h"
   ⋮
for (int i = 0; i < numElements; ++i)
{
  ;
  cout << myArray[i] << endl;
}
;
myArray[0] = myArray[1];
```

Again, you can see that we can turn out debugging output on and off at will.

One final refinement worth noting. When you have a large program with many source code files, it's easy to get confused about which debugging output lines are coming from where. C and C++ define two special macros, `__FILE__` will be replaced by the name of the file (in quotes) in which it lies, and `__LINE__` is replaced by the line number in which it occurs.[8] So we can rewrite `debug.h` like this:

```
#ifdef DEBUG

#define dbgprint(x) cerr << \#x << ": " << x << " in " << __FILE__ << ":" << __LINE__ << endl
#define dbgmsg(message) cerr << message " in " << __FILE__ << ":" << __LINE__ << endl

#else

#define dbgprint(x)
#define dbgmsg(message)

#endif
```

[8] In case it's not clear, each of these symbols has a pair of _ characters in front and another pair in back.

to get debugging output like:

```
i: 0 in myProgram.cpp:23
i: 1 in myProgram.cpp:23
i: 2 in myProgram.cpp:23
    ⋮
i: 125 in myProgram.cpp:23
Exited loop in myProgram.cpp:26
```

7.4.2. assertions

Sometimes, we can anticipate potential trouble spots as we are writing the code. Good programmers often engage in defensive programming, in which they introduce into their code special checks and actions just in case things don;t actually behave as expected.

One staple of defensive programming is the assertion, a boolean test that should be true if things are working as expected. The `assert` macro, defined in header file `<assert.h>` for C and `<cassert>` for C++, allows us to introduce assertions into our code so that the program stops with an infomrational message whenever the asserted condition fails.

For example, back when we were first writing `myProgram.cpp`, we might have anticipated trouble with that loop and written:

```
#include <cassert>
  ⋮
```

```
assert (numElements <= myArraySize);
for (int i = 0; i < numElements; ++i)
{
  cout << myArray[i] << endl;
}
myArray[0] = myArray[1];
```

Assertions are controlled by the compile-time symbol NDEBUG ("not debugging"). If NDEBUG is defined, then each assertion is compiled as a comment - it doesn't affect the actual program execution. But if NDEBUG is not defined, then the assertion gets translated to something along these lines (it varies slightly among different compilers):

```
#define assert(condition) if (!(condition)) {\
  cerr << "assertion failed at " << __FILE__ << ":" << __LINE__ \
      << endl; \
  abort(); \
}
```

so our sample code would become:

```
#include <cassert>
   ⋮
if (!(numElements <= myArraySize)) {\
  cerr << "assertion failed at " << __FILE__ << ":" << __LINE__ \
      << endl; \
  abort(); \
}
for (int i = 0; i < numElements; ++i)
{
  cout << myArray[i] << endl;
}
myArray[0] = myArray[1];
```

Unlike the kind of debugging out we have looked at earlier, assertions are silent unless things are going wrong anyway, so many programmers don't bother turning them off in the final submitted version unless the conditions being tested are complicated enough to noticeably slow the execution speed.

7.4.3. Automated Debuggers

When debugging output and assertions aren't convenient or you need more details about what's going on, it's time to look at automatic debuggers.

When compiling with `gcc` and `g++`, the `-g` option causes the compiler to emit information useful for a run-time debugger. The debugger of choice with these compilers is called `gdb`.

For Java programs, the same `-g` option causes the compiler to emit information useful for its run-time debugger, called `jdb`.

Debuggers like `gdb` and `jdb` can be especially useful in dealing with silent crashes, where you really don't know where in the program the crash occurred.

1. Look at the output produced before the crash. That can give you a clue as to where in the program you were when the crash occurred.

2. Run the program from within a debugger (`gdb` if you have compiled with `g++`). Don't worry about breakpoints or single-stepping or any of that stuff at first. Just run it.

   When the crash occurs, the debugger should tell you what line of code in what file eas being executed at the moment of the crash.

   Actually, it's not quite that simple. There's a good chance that the crash will occur on some line of code you didn't actually write yourself, deep inside some system library function that was called by some other system library function that was called by some other . . . until we finally get back to your own code. That crash occurred because you are using a function but passed it some data that was incorrect or corrupt in some way.

   Your debugger should let you view the entire runtime stack of calls that were in effect at the moment of the crash. (Use the command "backtrace" or "bt" in `gdb` to view the entire stack.) So you shoud be able to determine **where** the crash occurred. That's not as good as determining **why**, but it's a start.

3. Take a good look at the data being manipulated at the location of the crash. Are you using pointers?

Could some of them be null? Are you indexing into an array? Could your index value be too large or negative? Are you reading from a file? Could the file be at the end already, or might the data be in a different format than you expected?

If you used a debugger to find the crash locations, you can probably move up and down the stack (`gdb` commands "up" and "down") and to view the values of variables within each active call. This may give a clue about what was happening.

4. Form some hypotheses (take a guess) as to what was going on at the time of the crash. Then *test your hypothesis*! You can do this a number of ways:

   (a) Add debugging output. If you think one of your variables may have a bad or unanticipated value, print it out. Rerun the program and see if the value looks OK. E.g.,

   ```
   cerr << "x = " << x << "  y = " << y << endl;
   cerr << "myPointer = " << myPointer << endl;
   cerr << "*myPointer = " << *myPointer << endl;
   ```

   (b) Add an assertion to test for an OK value. E.g.,

   ```
   assert (myPointer != 0);
   ```

   Rerun the program and see if the assertion is violated.

   (c) In the debugger, set a breakpoint shortly before the crash location. Run the program and examine the values of the variables via the debugger interface. Single step toward the crash, watching for changes in the critical variables.

Once you have figured out what was the immediate cause of the crash, then you're ready for the really important part.

5. Try to determine the ultimate reason for the problem.

   Sometimes the actual problem is right where the crash occurs. Unfortunately, it's all to common for the real "bug" to have occurred much earlier during the execution. But once you know which data values are incorrect or corrupted, you can strart trying to reason backwards through your code to ask what could have caused that data to *become* incorrect.

   As you reason backwards, continue to form hypotheses about what the earlier cause might be, and keep testing those hypotheses as described in the prior step.

7.4.4. emacs debugging mode

The easiest way to run `gdb` and `jdb` is, again, from inside `emacs`. The reason for this is quite simple. `emacs` will synchronize a display of the source code with the current debugger state, so that as you use the debugger to move through the code, `emacs` will show you just where in the code you are.

Try creating a longer program in C or C++, and compile it to produce an executable program `foo`. From within `emacs`, look at one of the source code files for that program and then give the command `M-x gdb`.[9]

At the prompt "`Run gdb like this:`", type the program name `foo`. `emacs` will then launch `gdb`, and eventually you will get the prompt "`(gdb)`" in a window. You can now control `gdb` by typing

[9] For Java programs, use `M-x jdb`, instead.

commands into the `gdb` window. The most important commands are:

set args . . . If your program expects arguments on its command lane when it is invoked from the shell, list those arguments in this command before running the program.[10].

break *function* Sets a breakpoint at the entry to the named function (i.e., indicates that you want execution to pause upon entry to that function).

break *lineNumber* Sets a breakpoint at the indicated line number in the current source code file. You can execute this command in emacs either by typing it directly in to the debugger command window pane, or by changing to a window containing the source code, positioning the cursor on the desired line, and giving the emacs command `C-X spacebar`,

run Starts the program running.

c **C**ontinues execution after it has been paused at a breakpoint.

n Executes the **n**ext statement, then pauses execution. If that statement is a function/procedure call, the entire call is performed before pausing.

You can also do this by giving the `emacs` command `C-C C-N`.

s Like `n`, executes the next statement, but if that statement is a function procedure call, this commands **s**teps into the body of the function/procedure and pauses there.

You can also do this by giving the `emacs` command `C-C C-S`.

[10]These may include redirection of the input and output

bt (backtrace) prints a list of all active function calls from `main` down to the function where execution actually paused.

up Moves the current debugger position up the call stack, allowing you to examine the caller of the current procedure/function.

You can also do this via the `emacs` command `C-C <`

down (or just "d" for short) Moves the current debugger position down the call stack, reversing the effect of a previous "up" command.

You can also do this via the `emacs` command `C-C >`

allowing you to examine the caller of the current procedure/function.

p *expression* Prints the value of *expression*, which may include any variables that are visible at the current execution location.

quit Ends your `gdb` session.

`gdb` also has a `help` command, that supplies more details on its available commands.

For Java programs, the `jdb` commands are similar to those listed above, but vary in minor details (e.g., you must spell out the command word "print" instead of abbeviating it to "p"). The easiest thing to do is to stick to the emacs commands (e.g., `C-c n` rather than typing out "next") and let emacs deal with the minor differences between debuggers.

7.5. Program Development under X

OK, we've seen so far that you have a variety of powerful tools available for aiding program development in a telnet or similar character-oriented environment. It should come as no surprise that adding a windowing capability will make everything just a little bit easier.

7.5.1. Compiling

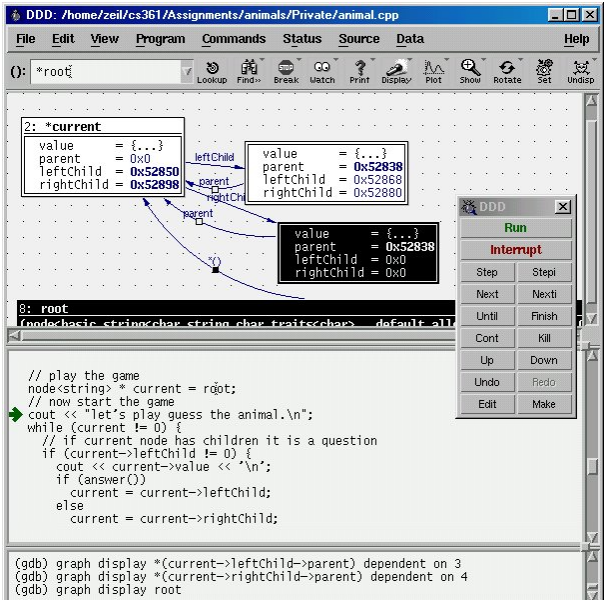
The basic tools and steps of compiling are largely unaffected by X. Compiling under `emacs` is still recommended, though `emacs` itself gains some helpful features when executed under X.

Particularly worth noting — the `emacs` menu bar, when running under X, adds C++-specific menus for editing and compiling source code.

7.5.2. Debugging

The `gdb` and `jdb` modes for controlling the debugger also acquire debugging-specific menus when `emacs` is run under X. Certainly, the `emacs` interface to the `gdb` and `jdb` debuggers remains a viable option when working under X.

You have another option, however, for an interface to `gdb` when running under X. `ddd`, the "Data Display Debuigger", is easier to use and offers a really nice display of data items with graphic depiction of pointer relationships (see Figure 7.5). You may, however, find it a bit sluggish on all but the fastest network connections.



(full size)

Figure 7.5: ddd, Showing Source Code and Data with Pointers

To get started using `ddd`, try working through the “Sample DDD Session” in the DDD manual.

Cygwin users: Although there is continued interest in getting `ddd` to work under CygWin, the versions I’ve tried have not been particularly stable. However, the CygWin port of `gdb` has its own built-in window interface, which comes up by default when you invoke `gdb`.

Chapter 8

Shell Games

8.1. Shore Are a Lot of Shells!

Although you may not have thought of it as such, the Unix commands that you type into your keyboard consititute a programming language. This language is interpreted by a program called a shell.

The Unix operating system is positively overrun with different shell languages. The C-shell (`csh`) or TC-shell (`tcsh`) that you have probably been using as your primary means of interaction is just one example. And, although we have not touched on it yet, that is a full programming language, including control flow instructions for branching and loops.

`csh` and `tcsh` constitute one family of commonly used shells. They languages they accept are nearly identical. `tcsh` adds features that are of especial use when typing commands directly to the shell. `csh` is preferred when writing scripts, short programs in the shell language, if only because `csh` is more likely to be installed on any given Unix system.

Another family is the `sh` family, which contains `sh`, `bsh`, and `bash`. `sh` is commonly used for writing scripts but lacks some of the conveniences we would want in an interactive shell. `bsh` is the "Bourne shell" (named for its inventor), and `bash`, the default interactive shell for Linux and CygWin, is the "Bourne Again shell".

These are by no means the only shell or scripting languages available for Unix, but they'll do for now.

What distinguishes these shell languages from more general-purpose programming languages is their emphasis on simplicity so as to permit apparently immediate translation. If, every time you typed a command, you had to wait while a full-fledged compiler was loaded and executed to process the characters you had typed, the time delays would have you screaming in frustration. Instead, these languages are designed to very quickly determine (usually by examining the first word you typed) determine if the

command is a special command “built in” to the shell. Anything else is a program to be launched.

The command `type` can be used to determine if a given word represents a built-in shell command, a program (and if so, where that program is located), or something else. Try logging in and giving the following commands:

```
type g++
type cp
type echo
type more
type if
type type
type foobar
```

We’ll concentrate on the `csh` for scripting purposes, with some notes on how `sh`-like languages, including `bash`, differ from `csh`.

8.2. Environment Variables

In any programming language, we expect to have variable names, and shells are no exception. A typical shell variable name begins with an alphabetic character and continues with zero or more alphanumeric characters. Oddly enough, when we want to get the value of a shell variable, we add a $ to the front of it.

Try this:

csh, tcsh	**sh, bash**
`set A=hello` `echo $A`	`A=hello` `echo $A`

and the `echo` command will produce the string “Hello”.

All shell variables hold strings, although in selected instances we may be able to interpret those strings as numbers.

The scope rules for shell variables are a bit odd. Most variables are local to the process where they are assigned. For example, look at the following command sequence:

csh, tcsh	sh, bash
`set A=hello`	`A=hello`
`echo $A`	`echo $A`
`hello`	`hello`
`csh`	`sh`
`echo $A`	`echo $A`
`A: undefined variable`	`A: undefined variable`
`exit`	`exit`
`echo $A`	`echo $A`
`hello`	`hello`

What's going on? The first `=` and `echo` are obvious enough. We set `A` to a value and then printed it. In the next line, we start another copy of the shell running (`csh` or `sh`). This runs as a separate process (the old one is temporarily suspended). This new "child" process does not inherit the variable `A`, so when we try to print it, we are told that `A` is undefined. The `exit` command shuts down the child process, returning us to the original (parent) process where we had previously defined `A`, and so we were able to print it again.

If we prefer, we can make a variable exported, meaning that its value will be seen by child processes. In `csh` and `tcsh` this is done by using `setenv` instead of `set`. In `sh` and `bash` this is done by naming the variable in an `export` command.

csh, tcsh	sh, bash

`setenv B goodbye`
`echo $B`
`goodbye`
`csh`
`echo $B`
`goodbye`
`exit`
`echo $B`
`goodbye`
	`B=goodbye`
	`export B`
	`echo $B`
	`goodbye`
	`csh`
	`echo $B`
	`goodbye`
	`exit`
	`echo $B`
	`goodbye`

An annoying inconsistency in the `csh` family: You must use an `=` sign when doing `set`:

```
set A=hello
```

but you must not use an `=` with `setenv`.

```
setenv B goodbye
```

(It only took me 15 years to get used to that!)

Many Unix programs use exported shell variables to control or modify their behavior. Examples that may be familiar include:

TERM Way back when we were learning to log in with telnet, we had to set the `TERM` variable to indicate what kind of terminal our telnet client was emulating. Many Unix programs use this setting to determine what control character sequences will work on our display.

DISPLAY The `DISPLAY` variable is used by X applications to determine where to send the windows and graphics for display to the person running the application.

Another important exported environment variable is `PATH`. This determines what programs you can execute without typing in a full path name. If you type a command that is not a built-in shell command, and does not contain a '/', then the shell looks at each directory listed in `$PATH` to see if the program you have requested can be found there. For example, suppose you have compiled a C++ program and produced a new program, `yourProgram`, in your current working directory. Some people will be able to execute the program this way:

```
yourProgram
```

while others will have to do it like this:

```
./yourProgram
```

The difference stems from whether your account has been set up so that your working directory (`.`) is in your `$PATH`. If it is, then you can use the first form. If it is not, you must use the second form.[1] To see your path, just give the command

[1] Some people consider it a bit of a security risk to have `.` in your `$PATH`, arguing that you could get spoofed into doing some very strange things if someone had deposited some malicious programs into your working directory under innocuous names like "`ls`" or "`cp`". My own feeling is that the threat here is pretty small.

```
echo $PATH
```

to print it, jsut as you would print any environment variable.

Adding additional directories to your $PATH is one of the more common customizations that people make to their Unix environments.

To see some of the other environment variables that are already defined for you, try giving the command env.

8.3. Scripts

You can put any sequence of Unix commands into a file and turn that file into a command. Such a file is called a *script*. For example, suppose that you are working on a program `myprog` and have several files of test data that you run through it each time you make a change. Create a file `dotest1` with the following lines:

```
./myprog < test1.dat > test1.dat.out
./myprog < test2.dat > test2.dat.out
./myprog < test3.dat > test3.dat.out
./myprog < test4.dat > test4.dat.out
./myprog < test5.dat > test5.dat.out
```

You can't execute `dotest1` yet, because you don't have execute permission. (Do `ls -l dotest1` to see this.) So use the `chmod` command to add execute permission:

```
chmod u+x dotest1
```

Now you can execute `dotest1` by simply typing

```
./dotest1
```

Most shells provide special facilities for use in scripts. Since these differ from one shell to another, it's a good idea to tell Unix which shell to use when running the script. You do this by placing the command `#!/bin/csh` in the first line of the script.

In fact, you can list any program there, not just `/bin/csh`, and Unix will use that program to process the remainder of the lines in the script.

- Any text file with execute permission can be invoked as a program.
- The first line must identify the program to be run (after a starting `#!`)
- The remaining lines are fed to the standard input of that program.

So another possibility for executing your test code would be to put the line `#!./myprog` at the front of each of the test data input files `test1.dat`, `test2.dat`, ..., and then execute those data files!

8.3.1. Parameters

We can pass parameters to shell scripts from the command line. For example, suppose we wanted a script to execute a single test like this:

```
dotest2 test1.dat
```

that would feed `test1.dat` (or whatever) to the input of `myprog`, saving the output in `test1.dat.out`.

In `csh`, we use the symbol `$k` to stand for the k^{th} argument given to the script. So we can write our script `dotest2`, as follows:

```
#!/bin/csh
./myprog < $1 > $1.out
```

After the appropriate `chmod`, this could then be invoked as

```
./dotest2 test1.dat test2.dat test3.dat test4.dat test5.dat
```

Of course, scripts can have more than one parameter. For example, try saving this file as `mcdonald`

```
#!/bin/sh
echo Old McDonald had a farm.
echo EIEIO
echo And on that farm he had a ‘$‘1.
echo EIEIO
echo With a ‘$‘2, ‘$‘2 here,
echo And a ‘$‘2, ‘$‘2 there.
```

and then invoking it this way:

```
./mcdonald cow moo
./mcdonald dog bark
```

8.3.2. Control Flow

Shells feature control flow based on “status codes” returned by programs. For example, this script tests to see if the `gcc` compiler has successfully compiled a file.

csh, tcsh	sh, bash

`if ( { gcc -c expr.c } ) then`
`  echo All is well.`
`else`
`  echo Did not compile!`
`endif`
	`if gcc -c expr.c;`
	`then`
	`  echo All is well.`
	`else`
	`  echo Did not compile!`
	`fi`

In `sh` (and its relatives `bsh` and `bash`), the `if` is followed by a list of commands, each terminated by a ';'. The status code of the last command is used as the `if` condition. In `csh` (and its relative, `tcsh`), the condition must be placed inside parentheses. To execute a command as part of the condition, place that inside {} brackets (with white space to either side of each bracket).

This idea of returning a status code explains why, in C and C++, the function `main` is always supposed to return an `int`. The returned value is the status code. Oddly enough, however, a return value of 0 is considered a success (taking the `then` branch in this example) while any non-zero value denotes a failure (the `else` branch).

Looping is also available in the shells. One of the most commonly used loop forms is that of visiting every item in some list. The list is often all files satisfying some wildcard pattern:

csh, tcsh	sh, bash

`foreach file (*.txt)`
`  echo $file`
`end`
	`for file in *.txt`
	`do`
	`  echo $file`
	`done`

Another common kind of loop visits every parameter passed to the script

csh, tcsh	sh, bash

`#!/usr/bin/csh`
`foreach file ($*)`
`  echo $file`
`end`
	`#!/usr/bin/sh`
	`for x in $*`
	`do`
	`  echo $x`
	`done`

`$*` is a list of all the command parameters. If either of these scripts were stored in a file `testp` and then invoked as

```
./testp a b c
```

the output would be

a
b
c

If invoked as

```
./testp *.txt
```

the output would be a list of all `.txt` files in the working directory.

Another common scripting pattern is a script that has few special parameters at the beginning, followed by an arbitrary number of remaining parameters (often filenames). The `shift` command helps us to handle these. Each call to `shift` removes he first element of the `$*` list. These scripts, stored as `testp2`:

csh, tcsh

```
#!/usr/bin/csh
set p1=$1
shift
set p2=$1
shift
echo The first two parameters are\
 $p1 and $p2
echo The remaining parameters are
foreach file ($*)
  echo $file
end
```

sh, bash

```
#!/usr/bin/sh
p1=$1
shift
p2=$1
shift
echo The first two parameters are\
 $p1 and $p2
echo The remaining parameters are
for x in $*
do
  echo $x
done
```

and invoked as

```
./testp2 a b c d
```

would produce the output

```
echo The first two parameters are a and b
echo The remaining parameters are
c
d
```

A while loop is also available.

csh, tcsh	sh, bash

`while ( condition )`
`  commands`
`end`
	`while condition;`
	`do`
	`  commands`
	`done`

but it's kind of hard to see just what you could do with it given the limited form of conditions we've looked at so far.

8.3.3. The `test` and `expr` Programs

One thing that becomes obvious quickly is that the status code based testing is limited. Often we want to

- Check for the presence/status of files
- Do tests on strings, especially in variables & script parameters

- Do numeric tests.

but status codes indicating whether or not a program executed successfully don't seem to help us very much.

The solution is to use a program whose job is to

- do the tests, and
- return the appropriate value as a status code.

The Unix program that does this is called `test`:

csh, tcsh

```
gcc -c expr.c
if (-r expr.o) then
  echo All is well.
else
  echo Did not compile!
endif
```

sh, bash

```
gcc -c expr.c
if test -r expr.o;
then
  echo All is well.
else
  echo Did not compile!
fi
```

The `csh` shells automatically call `test` in their `if` and `while` loop conditions. The `sh` family does not.

`test` takes a bewildering variety of possible parameters. You can see the whole list by giveng the command `man test`. Many of these are used for checking the status of various files. The `-r expr.o` in the above shells checks to see if a file named `expr.o` exists and if we have permission to read it. Some common file tests are:

test	is true if
`-r file`	*file* exists and is readable
`-w file`	*file* exists and is writable
`-x file`	*file* exists and is executable
`-d file`	*file* exists and is a directory

Strings are compared with `=~` and `!~` in `csh`, `=` and `!=` in `sh`.

csh, tcsh

```
if ( $USER =~ zeil ) then
  echo Nice guy!
else
  echo Who are you?
endif
```

sh, bash

```
if test $USER = zeil;
then
  echo Nice guy!
else
  echo Who are you?
fi
```

Numbers are compared with the conventional relational operators (`== != < > <= >=`) in `csh`, but have more clumsy equivalents in `sh` (`-eq -ne -lt -gt -le -ge`):

csh, tcsh	sh, bash

`if ( $count == 0 ) then`
`  echo zero`
`else`
`  echo non-zero`
`endif`
	`if test $count -eq 0;`
	`then`
	`  echo zero`
	`else`
	`  echo non-zero`
	`fi`

But where do numbers come from if all the variables contain strings?

From yet another program, of course. The shells themselves have no built-in numeric capability. Calculations can be performed by the `expr` program. This program treats its arguments as an arithmetic expression, evaluates that expression, and prints the result on standard output. For example:

```
expr 1
1
expr 2 + 3 \* 5
17
```

Note the use of \ to "quote" the following character (`*`). Without that backwards slash, the shell into which we typed the command would treat the `*` as the filename wildcard, replacing it with a list of all files in the current directory, and `expr` would have actually seen something along the lines of

```
expr 2 + 3 file1.txt file2.txt myfile.dat 5
```

Now, how do we get the output of an `expr` program evaluation into a variable or a script expression where it can do some good? For this we use the convention that the backwards apostrophes, ‘, when used to quote a string, mean “execute this string as a shell command and put its standard output right here in this command”.

For example:

```
echo Snow White and the expr 6 + 1 Dwarves
Snow White and the expr 6 + 1 Dwarves
echo Snow White and the ‘expr 6 + 1‘ Dwarves
Snow White and the 7 Dwarves
```

With these two ideas, we can now do numerical calculations in our scripts:

csh, tcsh

```
set count=0
foreach file (*)
  set count=‘expr $count + 1‘
end
echo There are $count files\
 in this directory.
```

sh, bash

```
count=0
for file in *
do
  count=‘expr $count + 1‘
done
echo There are $count files\
 in this directory.
```

8.3.4. Scripting Example: The Student Test Manager

As an example of how to bring all those scripting details together, let's look at some scripts to aid in testing programs. Many programming students wind up adopting a hit-and-miss approach to testing their code, in part because they don't set themselves up with an easy way to repeat a large number of tests every time they make a change to their programs.

What we'd like to end up with is a simple system for testing non-GUI programs. The idea is that the student designs a number of tests and can then issue a command to run all or a selected subset of those tests. The command should run those tests, capturing the program outputs in files that the student can examine later. Furthermore, we can save the student a bit of time by letting him or her know if the program output has changed on any of those tests.

Sample Scenario

So, suppose the student is working on a program named `myProg` and has designed 20 test cases. Once the program compiles successfully, the student could say

```
./runTests ./myProg 1 20
```

to run all 20 tests. The output might look something like:

```
Starting test 1...
Starting test 2...
```

```
Starting test 3...
Starting test 4...
```

Maybe at this point the output stops, suggesting that the program has been caught in an infinite loop. The student kills the program with ^-C. Now looking in the directory, the student finds files `testOut1.txt`, `testOut2.txt`, `testOut3.txt`, and `testOut4.txt` corresponding to the tests thatwere actually started. The student looks at the first 3 of these, decides that the captured output looks correct, and starts debugging the program to figure out why it hung up on test 4. Eventually the student makes change to the program, recompiles it, and tries again, this time just running test 4.

```
./runTests ./myProg 4 4
Starting test 4...
** Test 4 output has changed
```

Not surprisingly, the output from test 4 is different, because the infinite loop has now, apparently been fixed. Checking `testOut4.txt`, everything looks good.

Encouraged, the student launches the whole test set once again

```
./runTests ./myProg 1 20
Starting test 1...
Starting test 2...
** Test 2 output has changed
Starting test 3...
** Test 3 output has changed
```

```
Starting test 4...
Starting test 5...
  ⋮
Starting test 20...
```

The unexpected has occurred. The fix to make test 4 work has changed the behavior of the program on tests 2 and 3, which had previously been believed to be OK. The student must go back and check these (as well as the outputs of tests 5…20) to see what has happened. Annoying? Yes, but it's better that the student should discover these changes in behavior before submitting than that the grader should do so after submission!

The Script

As we have envisioned it, our `runTests` script takes three parameters:

1. The name of the program to run
2. The number of the first test to be performed
3. The number of the last test to be performed

So we start our script by gathering those three parameters:

csh, tcsh	sh, bash

`#!/usr/bin/csh`
`set programName=$1`
`set firstTest=$2`
`set lastTest=$3`
	`#!/usr/bin/sh`
	`programName=$1`
	`firstTest=$2`
	`lastTest=$3`

Clearly, the main control flow here will be a loop going through the requested test numbers.

csh, tcsh

```
#!/usr/bin/csh
set programName=$1
set firstTest=$2
set lastTest=$3
#
# Loop through all tests
set testNum=$firstTest
while ($testNum <= $lastTest)
   ⋮
  set testNum=‘expr $testNum + 1‘
end
```

sh, bash

```
#!/usr/bin/sh
programName=$1
firstTest=$2
lastTest=$3
#
# Loop through all tests
testNum=$firstTest
while test $testNum -le $lastTest;
do
   ⋮
  testNum=‘expr $testNum + 1‘
done
```

For each test, we will eventually want to compare the output from this test, stored in `testOut`*i*`.txt` to the output from the previous test, which we will assume is stored in `testOut`*i*`.old.txt`. The `cmp` command lets us compare two files to see if their contents are identical.

csh, tcsh

```
#!/usr/bin/csh
set programName=$1
set firstTest=$2
set lastTest=$3
#
# Loop through all tests
set testNum=$firstTest
while ($testNum <= $lastTest)
   ⋮
#
# Has the output changed?
  if (-r testOut$testNum.old.txt) then
    if ( ! { cmp testOut$testNum.old.txt testOut$testNum.txt } ) then
       echo \*\* Test $testNum output has changed
    endif
  endif
  set testNum=‘expr $testNum + 1‘
end
```

---

sh, **bash**

```
#!/usr/bin/sh
programName=$1
firstTest=$2
lastTest=$3
#
# Loop through all tests
testNum=$firstTest
while test $testNum -le $lastTest;
do
   ⋮
#
# Has the output changed?
  if test -r testOut$testNum.old.txt;
  then
    if cmp testOut$testNum.old.txt testOut$testNum.txt;
    then
       donothing=0
    else
       echo \*\* Test $testNum output has changed
    fi
  fi
  testNum=‘expr $testNum + 1‘
```

```
done
```

To set this up, we must determine just where the `.old.txt` files come from. They are simply the previous version of the test output files (if that particular test has ever been run).

csh, **tcsh**

```
#!/usr/bin/csh
set programName=$1
set firstTest=$2
set lastTest=$3
#
# Loop through all tests
set testNum=$firstTest
while ($testNum <= $lastTest)
#
# Save the previous output from this test
  if (-r testOut$testNum.txt) then
    /bin/mv testOut$testNum.txt testOut$testNum.old.txt
  endif
   ⋮
#
# Has the output changed?
```

```
  if (-r testOut$testNum.old.txt) then
    if ( ! { cmp testOut$testNum.old.txt testOut$testNum.txt } ) then
       echo \*\* Test $testNum output has changed
    endif
  endif
  set testNum=‘expr $testNum + 1‘
end
```

---

sh, bash

```
#!/usr/bin/sh
programName=$1
firstTest=$2
lastTest=$3
#
# Loop through all tests
testNum=$firstTest
while test $testNum -le $lastTest;
do
#
# Save the previous output from this test
```

```
  if test -r testOut$testNum.txt;
  then
    /bin/mv testOut$testNum.txt testOut$testNum.old.txt
  fi
   ⋮
#
# Has the output changed?
  if test -r testOut$testNum.old.txt;
  then
    if cmp testOut$testNum.old.txt testOut$testNum.txt;
    then
       donothing=0
    else
       echo \*\* Test $testNum output has changed
    fi
  fi
  testNum=‘expr $testNum + 1‘
done
```

Finally, we come to the heart of the matter. We need to actually execute the program, saving the output in the appropriate `testOut...` file. Exactly how we want to execute the program depends upon how the program gets its input data. I'm going to assume, for the moment, that this program reads its input data from the standard input stream, and that the student saves the input test cases in `testIn1.txt`,

testIn2.txt,....

csh, tcsh

```
#!/usr/bin/csh
set programName=$1
set firstTest=$2
set lastTest=$3
#
# Loop through all tests
set testNum=$firstTest
while ($testNum <= $lastTest)
#
# Save the previous output from this test
  if (-r testOut$testNum.txt) then
    /bin/mv testOut$testNum.txt testOut$testNum.old.txt
  endif
#
# Run the test!
  echo Starting test $testNum...
  $programName < testIn$testNum.txt >& testOut$testNum.txt
#
# Has the output changed?
```

```
  if (-r testOut$testNum.old.txt) then
    if ( ! { cmp testOut$testNum.old.txt testOut$testNum.txt } ) then
       echo \*\* Test $testNum output has changed
    endif
  endif
  set testNum=‘expr $testNum + 1‘
end
```

---

sh, bash

```
#!/usr/bin/sh
programName=$1
firstTest=$2
lastTest=$3
#
# Loop through all tests
testNum=$firstTest
while test $testNum -le $lastTest;
do
#
# Save the previous output from this test
```

```
  if test -r testOut$testNum.txt;
  then
    /bin/mv testOut$testNum.txt testOut$testNum.old.txt
  fi
#
# Run the test!
  echo Starting test $testNum...
  $programName < testIn$testNum.txt 2>&1 > testOut$testNum.txt
#
# Has the output changed?
  if test -r testOut$testNum.old.txt;
  then
    if cmp testOut$testNum.old.txt testOut$testNum.txt;
    then
       donothing=0
    else
       echo \*\* Test $testNum output has changed
    fi
  fi
  testNum=‘expr $testNum + 1‘
done
```

By making minor changes to the way the program is run, we can accommodate a number of different possible program styles. How would you change these scripts for a program that read no inputs at all, but could be invoked with different command-line parameters?

There are a number of possibilities, but I would put the various parameters into the `testIn...` files, and run them this way:

csh, **tcsh**

```
#!/usr/bin/csh
    ⋮
#
# Run the test!
  echo Starting test $testNum...
  $programName ‘sed -e s/[\\r\\n]//g testIn$testNum.txt‘ \
   >&  testOut$testNum.txt
    ⋮
```

---

sh, **bash**

```
#!/usr/bin/sh
    ⋮
```

```
#
# Run the test!
  echo Starting test $testNum...
  $programName ‘sed -e s/[\\r\\n]//g testIn$testNum.txt‘ \
   2>&1 > testOut$testNum.txt
    ⋮
```

8.4. Customizing Your Unix Environment

When a C-shell or TC-shell is started up (e.g., whenever you log in, or spawn off a new `xterm`), it executes the commands in a file called `~/.cshrc` This gives you an opportunity to customize your Unix environment.[2]

You may or may not already have a `.cshrc` file.[3]. You can check by giving the command

```
ls -a ~
```

If you don't have a `.cshrc` file, you should make one. If you do, consider changing it as described here. Edit your `.cshrc` file and insert the following:

```
setenv EDITOR emacs
umask 002
limit coredumpsize 0
#
#   skip remaining setup if not an interactive shell
#
if ($?USER == 0 || $?prompt == 0) exit
set history=40
set ignoreeof
```

[2]A related file is `~/.login`, which is only run when you first log in to the system. This can also be customized, but, on the whole, `~/.cshrc` is probably more useful.

[3]Note that because this filename starts with a ".", you won't see it with an `ls` command unless you use the `-a` option

```
set prompt="'hostname': "
alias cp            'cp -i'
alias mv            'mv -i'
alias rm            'rm -i'
alias ls            'ls -F'
alias ff           'find . -name \!* -print'
```

The `setenv` line indicates that `emacs` is your editor of choice. Some programs, including many e-mail programs, will use this information to load an editor when you have large amounts of text to enter/alter.

The `umask` command sets your default file protections. See this discussion for details.

Of the remaining lines, the most interesting are the `alias` commands. These set up "abbreviations" for commands. In this case, we are mainly adding options to familiar commands. The first three aliases add a `-i` option to the `cp`, `mv`, and `rm` commands. This option causes each command to prompt for a confirmation whenever its action would result in a file being deleted. The fourth alias adds the `-F` option to all `ls` commands, making it easier to tell the difference between ordinary files, directories, and executable programs. The final alias sets up a "find-file" command, `ff`. This will search the current directory and all subdirectories within it for a file matching a given pattern. For example the command sequence

```
cd ~
ff '*.txt'
```

will list all of your files with the `.txt` extension.

After you have checked this file and saved it, try invoking a new copy of the shell

```
tcsh
```

to test out the changes in behavior.

Another change worth considering in a `.cshrc` file is adding additional directories to your `PATH`. For example, if you want to be sure that `xterm` and other `X` client programs are readily available whenever you are logged in, you would want to make sure that /cd/usr/local/X11R6/bin is in your `$PATH`. You can do this by adding the command

```
set path = (/usr/local/X11R6/bin $path)
```

to your `.cshrc` file.

Chapter 9

Conclusion

9.1. Where to Go From Here?

We've only scratched the surface in this document. There are many more useful commands and programs available on the CS Department Unix machines, and many of the commands that we have covered have additional options that have not been mentioned here. Remember that you can use the Unix `man` command to call up documentation on any command. The appendix lists a number of additional commands that you may want to check out as you become more familiar with Unix.

Appendix A

Appendices

A.1. Unix Command Summary

[] denotes options
{} denotes required argument
^ denotes control key (depress while typing listed letter).
. . . indicates that command has many options. Use `man` to learn about this command.

awk . . .	a pattern matching and text manipulation language.
bg	puts process in background after ^z
cal [month] {year}	displays calendar for that month
cal	displays calendar for current month
cat {filename}	displays filename
cat [options]	
-b	number the lines, as -n, but omit the line numbers from blank lines.
-n	precede each line output with its line number.
cd [directoryname]	changes to directoryname, no argument indicates home directory
cd ..	changes to directory one above current
cp {file1} {file2}	copy file1 naming it file2
mv {file1} {file2 or directoryname}	move files or rename them
date	displays date
diff {file1} {file2}	compares two files, reporting any differences
echo	repeats line; useful when using $*$ and ? in filenames
fg	puts first command in background into the foreground
grep {pattern} {filename}	find pattern in filename
head -n	Prints the first n lines of its input, ignoring the rest

kill [option] {process id #}	stop a process
-9	kill no matter what: can be DANGEROUS
logout	end session, must be in login shell
lpr {filename}	send file to printer for printing
lprm {request} {userid}	remove a file from the printer queue
lpq	check status of printer and jobs
ls [options]	list files
-l	long form
-a	all files, including .files
-g	groups
mail	see ”man mail” and /home/public/help for more information
man [option] {command}	display manual page for command
mesg {y or n}	enable/disable messages to terminal
mkdir {directoryname}	create a directory
more {filename}	list filename one screen at a time
nroff,troff . . .	text formatting programs
ps	show processes you are running
pwd	print working directory
rm [option] {filename}	remove files
-i	interactive
-r	recursive (use with caution)

rmdir {directoryname}	remove directoryname
rwho	who is on your current network
X	X windows environment
openwin	openwindows environment
sed ...	A non-interactive editor, useful for writing scripts that involve string replacements, line deletions, etc.
sort [options] {filename}	sort filename
-b	ignore spaces and tabs
-f	sort upper- and lower-case together
-r	reverse the sorting order
-o filename	save the output of sort in filename
-t letter	set field separator to letter
-u	remove duplicate lines
spell {filename}	check spelling of filename
tail -n	Prints the last n lines of its input,
tr	Replaces/deletes characters ignoring the rest
wc [options] {filename}	count words, lines, and characters
-c	characters only
-l	number of lines only
-w	number of words only
who	who is running remote logins on your machine

write {user}	write message to user, ^d to quit
yppasswd	change password, follow prompts

?	matches any single character in a filename
*	matches any number of characters in a filename (or no characters)
&	puts command in background when appended to a command line
|	pipe, connects output of one command with input of another
>	redirects output of a command to a file, erasing current contents of a file
>>	appends output of a command to an existing file
<	uses the file as an input for a command
^c	aborts process (useful when ”hung-up”)
^d	stops a process or signals ”done” on console, indicates logout

A.2. Emacs Command Summary

EMACS command summary

A.3. Linking to this Document

Instructors interested in linking directly to specific sections of this document may do so by appending the appropriate anchor name to the URL of this document, e.g.,

```
http://www.cs.odu.edu/~cs252/Lectures/Unix/unix_w.pdf#loggingin
```

The defined anchors are listed in the tables below.

Anchor	Section	Section Title
basicUnix	2.4	Some Basic Unix Commands
catcmd	7	Some Basic Unix Commands
commands	A.1	Unix Command Summary
compshell	7.1.1	Compiling C and C++ in the Shell
controlflow	8.3.2	Control Flow
creatingsession	12	Starting StarNet X-Win32
custom	8.4	Customizing Your Unix Environment
debugging	7.4	Debugging
dostounix	6.1.4	Problems and Inconsistencies
emacs	3.1	Editing Text Files
emacscompile	7.2	Compiling in emacs
emacsx	4.2	Editing under X
envvars	8.2	Environment Variables

Anchor	Section	Section Title
fileprot	2.6	File Protection
filetransfer	6.1	File Transfer
firewall	4.1.4	Firewalls, NAT, & Internet Connection Sharing
forwarding	5.2	Forwarding Addresses
gccCompilation	7.1	Compilers
linking	A.3	Linking to this Document
loggingin	2.1	Logging In
lsF	7	Some Basic Unix Commands
lsa	7	Some Basic Unix Commands
mail	5.1	Using Electronic Mail
mailsend	5.1.3	Sending
make	7.3	Project Management with Make
pinecompose	5.2	Sending Messages
pineindex	5.3	Receiving Messages
pinemain	5.1	The PINE E-mail program
progdevx	7.5	Program Development under X
redirect	2.5	Redirection and Pipes
rlogin	7	Some Basic Unix Commands
shellscripts	8.3	Scripts

Anchor	Section	Section Title
telnet	2.1.1	Making a Connection: telnet
termtypes	2.1.3	Setting Your Terminal Type
theBasics	2	The Basics
typing	2.3	Typing Unix Commands
umask	2.6.3	Beware the umask!
unixFiles	2.2	The Unix File System
unixtodos	6.1.4	Problems and Inconsistencies
vim	3.1	Editing Text Files
vimcompile	7.2.1	Compiling with vim
xfermode	6.1.1	Before You Start: Understanding Transfer Modes
xwin	4.1	The X Window System